

"Paul Graham's prose is brimming with contrarian insight and practical wisdom."

—Andy Hertzfeld, co-creator of the Macintosh computer

PAUL GRAHAM

HACKERS & PAINTERS

BIG IDEAS FROM THE COMPUTER AGE



PREFACE

はじめに

This book is an attempt to explain to the world at large what goes on in the world of computers. So it's not just for programmers. For example, Chapter 6 is about how to get rich. I believe this is a topic of general interest.

本書は、コンピュータの世界で今何が起きているかを、より広い世界に向けて説明しようとする試みだ。プログラマのためだけの本ではない。例えば第 6 章では、どうしたら裕福になれるかを論じている。一般の人々にとってもこれは興味ある話題だろう。

You may have noticed that a lot of the people getting rich in the last thirty years have been programmers. Bill Gates, Steve Jobs, Larry Ellison. Why? Why programmers, rather than civil engineers or photographers or actuaries? "How to Make Wealth" explains why.

過去 30 年ほどの間に裕福になった人々の多くがプログラマであった、ということに気付いただろうか。ビル・ゲイツ、スティーブ・ジョブズ、ラリー・エリソン。ふむ。どうしてプログラマなんだろう？土木技師や写真家や保険数理士ではなくて。「第 6 章 富の創りかた」ではこれを説明してみた。

The money in software is one instance of a more general trend, and that trend is the theme of this book. This is a Computer Age. It was supposed to be the Space Age, or the Atomic Age. But those were just names invented by PR people. Computers have had far more effect on the form of our lives than space travel or nuclear technology.

ソフトウェアにおける金は、もっと一般的な変化のひとつの表れにすぎない。本書のテーマはその変化だ。今、私たちはコンピュータの世界に生きている。かつて未来は宇宙の時代、あるいは原子力の時代になると言われていた。でもこういったものは広告部門がひねりだした名前にすぎない。コンピュータは、宇宙旅行や核技術よりもずっと大きな影響を私たちの生活に与えている。

Everything around us is turning into computers. Your typewriter is gone, replaced by a computer. Your phone has turned into one. So has your camera. Soon your TV will. Your car has more processing power in it than a room-sized mainframe had in 1970. Letters, encyclopedias, newspapers, and even your local store are being replaced by the Internet. So if you want to understand where we are, and where we're going, it will help if you understand what's going on inside the heads of

hackers.

身の回りの全てのものがコンピュータへと変わりつつある。タイプライターは消え、コンピュータになった。電話もコンピュータになった。カメラもそう。テレビもすぐにそうなるだろう。自家用車に入っているコンピュータは、1970年に一部屋まるまる占めていたメインフレームよりも強力だ。手紙も百科事典も新聞も、そして身近な小売商店さえも、インターネットで置き換えられようとしている。だから、今私たちがどこにいてどこへ向かうとしているのかを理解したかったら、ハッカーの頭の中で起こっていることを知るのが役に立つ。

Hackers? Aren't those the people who break into computers? Among outsiders, that's what the word means. But within the computer world, expert programmers refer to themselves as hackers. And since the purpose of this book is to explain how things really are in our world, I decided it was worth the risk to use the words we use.

え、ハッカーだって？それってコンピュータに侵入する連中のことじゃないの？うん、外部の人にとっては、ハッカーという言葉はそういう意味だ。けれどコンピュータの世界では、優れたプログラマたちが、やはり優れたプログラマのことを指してハッカーと呼ぶんだ。この本の目的はコンピュータの世界が本当はどうなっているのかを説明するってことだから、誤解される危険を承知の上で、敢えてこの単語、私たちが実際に使っている単語を使うことにしたんだ。

The earlier chapters answer questions we have probably all thought about. What makes a startup succeed? Will technology create a gap between those who understand it and those who don't? What do programmers do? Why do kids who can't master high school end up as some of the most powerful people in the world? Will Microsoft take over the Internet? What to do about spam?

前半の章では、たぶんみんなが一度は考えたことがある疑問に対して答えを提示する。ベンチャー企業が成功する秘密とは。技術はそれを理解する者と理解しない者の間に格差を作るか。プログラマはいったい何をやっているの。高校の落ちこぼれが世界で最も力を持つ者になることがあるのはなぜだろう。マイクロソフトはインターネットを制覇するか。スパムにどう対応したらいい？

Several later chapters are about something most people outside the computer world haven't thought about: programming languages. Why should you care about programming languages? Because if you want to understand hacking, this is the thread to follow—just as, if you wanted to understand the technology of

1880, steam engines were the thread to follow.

しかし後半の章は、コンピュータの世界の外にいる人がたぶん考えたこともないようなことを扱う。プログラミング言語だ。なんでプログラミング言語のことを知らなくちゃならないんだって?ハッキングについて理解しようと思ったら、それが手がかりになるからだ。例えば 1880 年の技術について理解しようと思ったら、蒸気機関が鍵になるのと同じように。

Computer programs are all just text. And the language you choose determines what you can say. Programming languages are what programmers think in.

コンピュータのプログラムは単なる言葉の羅列だ。そして選ぶ言語によって何を言えるかが決まってくる。プログラミング言語はプログラマがものを考えるときに使う言葉なんだ。

Naturally, this has a big effect on the kind of thoughts they have. And you can see it in the software they write. Orbitz, the travel web site, managed to break into a market dominated by two very formidable competitors: Sabre, who owned electronic reservations for decades, and Microsoft. How on earth did Orbitz pull this off? Largely by using a better programming language.

必然的に、プログラミング言語の選択はプログラマの考えることに大きな影響を及ぼす。プログラマが書くソフトウェアにそれを見て取ることができる。旅行予約の Web サイトである Orbitz は、既に 2 つの大きな会社——オンライン予約システムを数十年にわたって提供していた Sabre と、マイクロソフト——によって牛耳られていたマーケットに割って入り、成功を収めた。一体全体どうやって Orbitz はそれを成し遂げたんだろう。より良いプログラミング言語を使うことによってだ。

Programmers tend to be divided into tribes by the languages they use. More even than by the kinds of programs they write. And so it's considered bad manners to say that one language is better than another. But no language designer can afford to believe this polite fiction. What I have to say about programming languages may upset a lot of people, but I think there is no better way to understand hacking.

プログラマは使う言語によっていくつかの部族に分かれる。どんなプログラムを書くかよりも、どの言語を使うかのほうが重要でさえある。だから、ひとつの言語が他の言語よりも良いと主張することは、礼儀正しいことではないとされてきた。でも言語を設計する者は、この上品な建前を信じているわけにはいかない。プログラミング言語に関して私が言わなければならないことは、たぶんたくさんの人々を怒らせるだろうけれど、ハッキングについて理解するのにこれ以上の方法はないと思うんだ。

Some might wonder about "What You Can't Say" (Chapter 3). What does that have to do with computers? The fact is, hackers are obsessed with free speech. Slashdot, the New York Times of hacking, has a whole section about it. I think most Slashdot readers take this for granted. But Plane & Pilot doesn't have a section about free speech.

「口にできないこと」(第 3 章)がコンピュータにどう関係があるんだ、と不思議に思う人がいるかもしれない。実はハッカーというのは、言論の自由に対してものすごく執着するものなんだ。ハッカーのニューヨークタイムズとも言えるスラッシュドットでは、言論の自由にひとつのセクションを割いている。スラッシュドットの読者のほとんどは、それを当然のことと受け止めているだろう。でも、例えば雑誌『Plane & Pilot』に言論の自由のセクションなどありはしない。

Why do hackers care so much about free speech? Partly, I think, because innovation is so important in software, and innovation and heresy are practically the same thing. Good hackers develop a habit of questioning everything. You have to when you work on machines made of words that are as complex as a mechanical watch and a thousand times the size.

どうしてハッカーは言論の自由を気にするんだろう。私が考えるひとつの理由は、ソフトウェアの世界では革新が何よりも重要であり、革新と異端は実質的には同じことだからだ。良いハッカーは全てのことを疑問に思う習慣を身に付けている。機械式時計のように複雑で、その 1000 倍もの部品でできた言葉の機械を相手にしていたら、そうならざるを得ない。

But I think that misfits and iconoclasts are also more likely to become hackers. The computer world is like an intellectual Wild West, where you can think anything you want, if you're willing to risk the consequences.

もともと、社会にうまく適合できなかったり偶像崇拜を嫌うものが、よりハッカーになりやすいということはあると思う。コンピュータの世界はいわば知的な「未開の西部」だ。結果を引き受ける気さえあれば、何でも考えることができる。

And this book, if I've done what I intended, is an intellectual Western. I wouldn't want you to read it in a spirit of duty, thinking, "Well, these nerds do seem to be taking over the world. I suppose I'd better understand what they're doing, so I'm not blindsided by whatever they cook up next." If you like ideas, this book ought to be fun. Though hackers generally look dull on the outside, the insides of their

heads are surprisingly interesting places.

そしてこの本は、私の目論見が成功していれば、知的な西部劇だ。「ふん、コンピュータオタクたちが世界を征服しかねない。やつらが何をしてるかを理解しておいて、次にやつらがでっちあげるものにだまされないようにしなくちゃ」といった義務感からこの本を読まないでほしい。新しいアイデアが好きなら、この本は楽しめるんじゃないかと思う。ハッカーは見かけは野暮ったいことが多いけれど、その頭の中はびっくりするほど面白い場所なんだ。

Cambridge, Massachusetts

マサチューセッツ州ケンブリッジにて

April 2004

TABLE OF CONTENTS

目次

Chapter 0 MADE IN USA	1
第0章 メイド・イン・USA.....	1
Chapter 1 WHY NERDS ARE UNPOPULAR	13
第1章 どうしてオタクはもてないか.....	13
Chapter 2 HACKERS AND PAINTERS.....	41
第2章 ハッカーと画家	41
Chapter 3 WHAT YOU CAN'T SAY.....	71
第3章 口にできないこと	71
Chapter 4 THE WORD "HACKER"	103
第4章 天邪鬼の価値	103
Chapter 5 THE OTHER ROAD AHEAD.....	114
第5章 もうひとつの未来への道.....	114
Chapter 6 HOW TO MAKE WEALTH	174
第6章 富の創りかた	174
Chapter 7 MIND THE GAP	215
第7章 格差を考える	215
Chapter 8 A PLAN FOR SPAM.....	242
第8章 スпамへの対策	242
Chapter 9 TASTE FOR MAKERS	269
第9章 ものづくりのセンス	269
Chapter 10 PLOGRAMMING LANGUAGES EXPLAINED	298
第10章 プログラミング言語入門.....	298
Chapter 11 THE HUNDRET-YEAR LANGUAGE	312
第11章 百年の言語.....	312
Chapter 12 BEATING THE AVERAGES	336
第12章 普通のやつらの上を行け.....	336
Chapter 13 REVENGE OF THE NERDS	357
第13章 オタク野郎の復讐.....	357
Chapter 14 THE DREAM LANGUAGE (BEING POPULAR)	389
第14章 夢の言語 (人気の言語を作るには).....	389
Chapter 15 DESIGN AND RESEARCH.....	426
第15章 デザインとリサーチ	426
Chapter 16 GREAT HACKERS	439

第 16 章 素晴らしきハッカー	439

Chapter 0 MADE IN USA

第0章 メイド・イン・USA

November 2004



(This is a new essay for the Japanese edition of [Hackers & Painters](#). It tries to explain why Americans make some things well and others badly.)

(これは、日本語版「ハッカーと画家」のための新しいエッセイである。米国人はなぜあるものを作るのは上手で、あるものを作るのは下手かを説明している。)

A few years ago an Italian friend of mine travelled by train from Boston to Providence. She had only been in America for a couple weeks and hadn't seen much of the country yet. She arrived looking astonished. "It's so *ugly!*"

数年前、イタリアの友人が列車でボストンからプロヴィンスまで旅行したことがある。彼女は米国に来てまだ 2 週間で、それまで米国をあまり見て回っていなかった。彼女は着くなり目を丸くしてこう言ったものだ。「なんて醜いんでしょう」

People from other rich countries can scarcely imagine the squalor of the man-made bits of America. In travel books they show you mostly natural environments: the Grand Canyon, whitewater rafting, horses in a field. If you see pictures with man-made things in them, it will be either a view of the New York skyline shot from a discreet distance, or a carefully cropped image of a seacoast town in Maine.

他の裕福な国から来た人々は、米国の人工物の粗雑さを想像もできないらしい。旅行ガイドには、グランド・キャニオン、ボートによる急流下り、牧場に散らばる馬といった、自然の風景が踊っている。人工物といえば、遠くから写したニューヨークのスカイラインか、巧みに切り取られたメイン州の海辺の町くらいのものだ。

How can it be, visitors must wonder. How can the richest country in the world look like this?

旅行者は不思議に思わずにはいられないだろう。なぜ世界で最も裕福な国が、こんなふう
に醜いんだ。

Oddly enough, it may not be a coincidence. Americans are good at some things and bad at others. We're good at making movies and software, and bad at making cars and cities. And I think we may be good at what we're good at for the same reason we're bad at what we're bad at. We're impatient. In America, if you want to do something, you don't worry that it might come out badly, or upset delicate social balances, or that people might think you're getting above yourself. If you want to do something, as Nike says, *just do it*.

奇妙なことだが、これは偶然ではないかもしれない。米国人には得意なこともあるが、苦手なこともある。映画やソフトウェアを作るのは得意だけれど、車や都市を作るのは下手くそだ。そして私が思うに、米国人は作るのが得意なものが得意である理由と、苦手なものが苦手である理由とは、同じなんだ。米国人は一度何かしたいと思い立つと、それがうまくできないんじゃないかとか、世間的に角が立つんじゃないかとか、あるいは無謀な試みだと人に思われるんじゃないかとか、そういうことを気にしない。何かしたくなったら、ナイキのコマーシャルみたいにするんだ。「とにかく、やる」。

This works well in some fields and badly in others. I suspect it works in movies and software because they're both messy processes. "Systematic" is the last word I'd use to describe the way [good programmers](#) write software. Code is not something they assemble painstakingly after careful planning, like the pyramids. It's something they plunge into, working fast and constantly changing their minds, like a charcoal sketch.

この方法がうまくゆく分野もあれば、そうでない分野もある。映画やソフトウェアはうまくゆくほうだろう。というのも、どちらもその製作過程はこんがらがったものだからだ。「几帳面」なんて言葉は、優れたプログラマがソフトウェアを書く仕事ぶりからは最も無縁なものだ。コードは、ピラミッドみたいに、慎重な計画をしてから苦勞して組み立てていくものじゃない。一気に集中して素早く手を動かしながら、常に気を変えていく、木炭スケッチみたいなものだ。

In software, paradoxical as it sounds, good craftsmanship means working fast. If you work slowly and meticulously, you merely end up with a very fine implementation of your initial, mistaken idea. Working slowly and meticulously is

premature optimization. Better to get a prototype done fast, and see what new ideas it gives you.

矛盾しているように聞こえるかもしれないが、ソフトウェアでは腕の良さは素早い仕事を意味する。ゆっくりと念入りに仕事をしていると、出来上がるものは当初のアイデアを精密に実現したものになるだろう。ただし、そのアイデアは間違っているだろうけどね。遅く念入りの仕事は早過ぎる最適化だ。むしろプロトタイプを素早く作り上げて、それによって新しいアイデアを得てゆくほうがよい。

It sounds like making movies works a lot like making software. Every movie is a Frankenstein, full of imperfections and usually quite different from what was originally envisioned. But interesting, and finished fairly quickly.

どの映画も一種のフランケンシュタインだ。欠陥は一杯だし、最初に思い描いていたものからは程遠い。でも面白いし、素早く作られる。

I think we get away with this in movies and software because they're both malleable mediums. Boldness pays. And if at the last minute two parts don't quite fit, you can figure out some hack that will at least conceal the problem.

映画やソフトウェアでそれが可能なのは、どちらも柔らかいメディアだからだ。そこでは大胆さが報われる。そして、最後の段階でどうしても噛み合わない要素が出てきたとしても、少なくとも何かのハックでそれを隠すことはできるだろう。

Not so with cars, or cities. They are all too physical. If the car business worked like software or movies, you'd surpass your competitors by making a car that weighed only fifty pounds, or folded up to the size of a motorcycle when you wanted to park it. But with physical products there are more constraints. You don't win by dramatic innovations so much as by good taste and attention to detail.

車や都市はそうはいかない。あまりに物質的すぎるからだ。車がソフトウェアや映画のようなものだったら、20kgしかない車だとか、駐車する時にバイクくらいの大きさにまで畳める車だとかを開発してライバルを出し抜くことだってできただろう。でも、物質的な製品というのは、もっとずっと制約の多いものだ。劇的な技術革新よりも、良いセンスと細部への注意が勝利を収めるものだ。

The trouble is, the very word "taste" sounds slightly ridiculous to American ears. It seems pretentious, or frivolous, or even effeminate. Blue staters think it's "subjective," and red staters think it's for sissies. So anyone in America who really cares about design will be sailing upwind.

問題はこの「センス」という言葉が、米国人には若干馬鹿げたものと思われていることだ。そういう言葉を使っていると、うぬぼれ屋か、軽薄か、あるいは軟弱であるとさえ思われる。進歩派からは「主観的だ」と思われ、保守派からは「女々しい」と思われる。だから米国で本当にデザインの問題を考えたい人は、逆風の中を進むことになる。

Twenty years ago we used to hear that the problem with the US car industry was the workers. We don't hear that any more now that Japanese companies are building cars in the US. The problem with American cars is bad design. You can see that just by looking at them.

20年前には、米国の自動車産業の問題は従業員の質だと言われていた。今はもうそんな話は聞かれない。日本の会社が米国で車を生産するようになったからだ。米国車の問題はデザインの悪さだ。見るだけでそれが分かるだろう。

All that extra sheet metal on the [AMC Matador](#) wasn't added by the workers. The problem with this car, as with American cars today, is that it was designed by marketing people instead of designers.

AMC マタドールにごてごて付いている余分な板金は、従業員が勝手に付け足したわけじゃない。この車の問題点は、今日の米国車の問題と同様、デザイナーではなく広告部門の連中によってデザインされたということだ。



1976 AMC MATADOR COUPE

Why do the Japanese make better cars than us? Some say it's because their culture encourages cooperation. That may come into it. But in this case it seems more to the point that their culture prizes design and craftsmanship.

なぜ日本人は米国人よりも良い車を作るんだろう。彼らの文化が協調作業を推奨している

からだと言う人もいる。それも要素のひとつではあるだろう。だが車に関しては、むしろ彼らの文化に、デザインと職人の仕事を尊ぶところがあることが重要なのだと思われる。

For centuries the Japanese have made finer things than we have in the West. When you look at swords they made in 1200, you just can't believe the date on the label is right. Presumably their cars fit together more precisely than ours for the same reason their joinery always has. They're obsessed with making things well. 幾世紀にもわたって、日本人は私たち西洋人よりも精緻なものを作ってきた。1200 年に作られた日本刀は、その時代に作られたことが信じられないほどだ。おそらく、日本車が米国人よりきっちりしているのは、日本できちんとした家具が作られてきたことと同じ理由による。彼らは、ものをうまく作ることに憑りつかれているんだ。

Not us. When we make something in America, our aim is just to get the job done. Once we reach that point, we take one of two routes. We can stop there, and have something crude but serviceable, like a Vise-grip. Or we can improve it, which usually means encrusting it with gratuitous ornament. When we want to make a car "better," we stick [tail fins](#) on it, or make it [longer](#), or make the [windows smaller](#), depending on the current fashion.

私たち米国人は違う。何かを作るとき、米国人はとにかく仕事を終えることを考える。とりあえず動くものができたら、そこからは 2 通りの道がある。そこで作るのをやめて、バイスグリップみたいに不格好だが何とか使えるものを使っていくか、あるいはそれを改善してゆく——でもたいてい、そういう改善とはごてごてした装飾を付けていくことだ。車を良くしようと米国人が考えるのは、その時々流行によって、尾ひれを付けてみたり、車体を長く伸ばしてみたり、窓を小さくしてみたりといったことにすぎない。



1959 CADILLAC ELDORADO



1975 CADILLAC ELDORADO



2004 DODGE MAGNUM

They still haven't learned.

Ditto for houses. In America you can have either a flimsy box banged together out of two by fours and drywall, or a McMansion-- a flimsy box banged together out of two by fours and drywall, but larger, more dramatic-looking, and full of expensive fittings. Rich people don't get better design or craftsmanship; they just get a larger, more conspicuous version of the standard house.

家もそうだ。米国で家と云ったら 2 種類しかない。ツーバイフォーの柱に石膏ボードを打ち付けたぺらぺらの箱か、あるいは超豪邸、つまり大きくてドラマチックな外観を備え、高価な家具を配置したツーバイフォーと石膏ボードのぺらぺらの箱だ。金持ちだからって、より良いデザインや優れた職人の仕事をもらえるわけじゃない。ただ、普通より大きくて目立つ家が手に入るだけだ。

We don't especially prize design or craftsmanship here. What we like is speed, and we're willing to do something in an ugly way to get it done fast. In some fields, like

software or movies, this is a net win.

米国では、良いデザインや職人の仕事は特に評価されない。私たちが好むのはスピードだ。速くできるなら醜い方法も喜んで許容する。ソフトウェアや映画などの分野では、これはトータルで有利な方向に働く。

But it's not just that software and movies are malleable mediums. In those businesses, the designers (though they're not generally called that) have more power. Software companies, at least successful ones, tend to be run by programmers. And in the film industry, though producers may second-guess directors, the director controls most of what appears on the screen. And so American software and movies, and Japanese cars, all have this in common: the people in charge care about design-- the former because the designers are in charge, and the latter because the whole culture cares about design.

これは、ソフトウェアや映画が柔軟なメディアであるというためだけじゃない。これらの業界では、デザイナーがより大きな力を持っている(一般的に「デザイナー」とは呼ばれていないかもしれないが)。ソフトウェア会社は、少なくとも成功しているところは、プログラマにより運営されていることが多い。映画業界では、プロデューサーが後ろについてあれこれ口を出しているとしても、たいていは監督が画面に現れるもののほとんどを決めている。したがって、米国のソフトウェアと映画、そして日本の車には共通点があるんだ。責任者がデザインに気を遣っているということだ。米国ではデザイナーが責任者となることで、日本ではその文化がデザインを重んじることで。

I think most Japanese executives would be horrified at the idea of making a bad car. Whereas American executives, in their hearts, still believe the most important thing about a car is the image it projects. Make a good car? What's "good?" It's so *subjective*. If you want to know how to design a car, ask a focus group.

だぶん、日本の自動車会社の経営者は、悪い車を生産するとかえただけでぞっとするんじゃないだろうか。だが米国の経営者たちは、心の底では、車で一番重要なのはそれが象徴するイメージだとまだ信じている。良い車だって? 「良い」って何のことだ? そんなの主観にすぎない。車のデザインについて知りたければ、フォーカスグループに聞けばいいじゃないか。

Instead of relying on their own internal design compass (like Henry Ford did), American car companies try to make what marketing people think consumers want. But it isn't working. American cars continue to lose market share. And the reason is that the customer doesn't want what he thinks he wants.

ヘンリー・フォードのように自分の内なるデザインの指針に頼るのではなく、米国の自動車会社は、消費者が望んでいると広告部門の連中が考えているものを作ろうとする。でもそれはうまくいっていない。米国車はマーケットシェアを減らし続けている。その理由は、消費者というのは自分が欲しいと考えているものを本当に欲してはいないからだ。

Letting focus groups design your cars for you only wins in the short term. In the long term, it pays to bet on good design. The focus group may say they want the meretricious feature du jour, but what they want even more is to imitate sophisticated buyers, and they, though a small minority, really do care about good design. Eventually the pimps and drug dealers notice that the doctors and lawyers have switched from Cadillac to Lexus, and do the same.

フォーカスグループに車をデザインさせるのは、短期的には成功しない。長期的には良いデザインに賭けるほうが勝つ。フォーカスグループはその時流行りのうわべの装飾を欲しがるともかもしれない。だが本当に欲しいのは、洗練された買い手の真似をすることだ。そして洗練された買い手は、数こそ多くないが、良いデザインに深くこだわる。売春婦のヒモや薬物密売人だって、医者や弁護士がキャデラックからレクサスに変えたことにいずれ気が付いて、真似をし始めるだろう。

Apple is an interesting counterexample to the general American trend. If you want to buy a nice CD player, you'll probably buy a Japanese one. But if you want to buy an MP3 player, you'll probably buy an iPod. What happened? Why doesn't Sony dominate MP3 players? Because Apple is in the consumer electronics business now, and unlike other American companies, they're obsessed with good design. Or more precisely, their CEO is.

アップルはこの米国の風潮のひとつの面白い反例だ。米軍で良い CD プレーヤーを買いたければ、たぶん日本製が選ばれるだろう。でも MP3 プレーヤーなら iPod だ。何が起こったんだろう。どうしてソニーが MP3 プレーヤー市場を独占していないんだ。アップルがコンシューマーエレクトロニクスに進出したからだ。他の米国の企業と違って、アップルは良いデザインに固執する。正確に言えば、アップルの CEO がそうなんだ。

I just got an iPod, and it's not just nice. It's *surprisingly* nice. For it to surprise me, it must be satisfying expectations I didn't know I had. No focus group is going to discover those. Only a great designer can.

私も最近 iPod を手に入れた。それは、ただ良いというだけじゃない。びっくりするほど良いんだ。何故びっくりしたかというと、それは自分でも気が付いていなかった期待を満たしてくれたからだ。フォーカスグループじゃそういう期待を見つけることはできない。た

だ、優れたデザイナーだけに可能なことだ。

Cars aren't the worst thing we make in America. Where the just-do-it model fails most dramatically is in our cities-- or rather, [exurbs](#). If real estate developers operated on a large enough scale, if they built whole towns, market forces would compel them to build towns that didn't suck. But they only build a couple office buildings or suburban streets at a time, and the result is so depressing that the inhabitants consider it a great treat to fly to Europe and spend a couple weeks living what is, for people there, just everyday life. [\[1\]](#)

車だけが、米国で作られる悪いものじゃない。とにかくやる、という方法が一番裏目に出るのが、都市だ。いやむしろ郊外都市と言おうか。もし不動産開発会社が十分に大規模で、街全てを自分で建てられるのだとしたら、市場の圧力からそれを良いものにしようとするだろう。だが普通はオフィスビルをあっちに 2 つ、郊外住宅地をこっちにひとつ、という具合に作るものだから、出来上がりはひどいものになる。そこの住人は、ヨーロッパに脱出してヨーロッパ人の日常生活を 2 週間ほど体験することが素晴らしいと思えるほどだ。

[\[1\]](#)



NOMINALLY DENVER, 2004

(Note new streets being built at lower left.)

But the just-do-it model does have advantages. It seems the clear winner for generating wealth and technical innovations (which are practically the same thing). I think speed is the reason. It's hard to create wealth by making a commodity. The real value is in things that are new, and if you want to be the first to make something, it helps to work fast. For better or worse, the just-do-it model is fast, whether you're Dan Bricklin writing the prototype of VisiCalc in a weekend, or a real estate developer building a block of shoddy condos in a month.

とにかくやる、という方法には、確かに利点がある。富を創りだし、技術を革新させる(両者は現実的には同じことだ)にはこれに勝る方法はないようだ。速度がその理由だと思う。だれでも作れる日用品を作ることで富を成すのは難しい。価値は新しいものにある。そして最初に何かを作り出したいと思うなら、速くやることは重要だ。良きにつけ悪しきにつけ、とにかくやる、という方法は速い。ダン・ブルックリンが VisiCalc のプロトタイプを週末に書き上げた時も、不動産会社が見かけ倒しのマンションの並ぶ街区をひと月ででっち上げる時も。

If I had to choose between the just-do-it model and the careful model, I'd probably choose just-do-it. But do we have to choose? Could we have it both ways? Could Americans have nice places to live without undermining the impatient, individualistic spirit that makes us good at software? Could other countries introduce more individualism into their technology companies and research labs without having it metastasize as strip malls? I'm optimistic. It's harder to say about other countries, but in the US, at least, I think we can have both.

「とにかくやる」方法と、注意深く進める方法のどちらかを選ばなければならないのだとしたら、私ならとにかくやるほうを選ぶだろう。でも、本当に選ばなくちゃならないんだろうか。両方を同時にやることはできないだろうか。米国人は、ソフトウェアをうまく作れるあの短期で独立心に富む精神を保ったまま、良い住環境を作ることができないんだろうか。他の国で、醜いショッピングモールをはびこらせることなく、技術企業や研究所にもっと個人主義を持ち込むことはできないんだろうか。私は楽観的だ。他の国について予測するのは難しいが、少なくとも米国では、私たちは両方を得ることが可能だと思う。

Apple is an encouraging example. They've managed to preserve enough of the impatient, hackerly spirit you need to write software. And yet when you pick up a

new Apple laptop, well, it doesn't seem American. It's too perfect. It seems as if it must have been made by a Swedish or a Japanese company.

アップルは元気づけられる例だ。ソフトウェアを書くのに必要な、短期でハッカー的な精神を十分に残していながら、アップルのノート PC を手にすると、それは米国的には感じられない。米国製にしてはうまくでき過ぎている。まるでスウェーデンか日本の企業が作ったみたいに。

In many technologies, version 2 has higher resolution. Why not in design generally? I think we'll gradually see national characters superseded by occupational characters: hackers in Japan will be allowed to behave with a [willfulness](#) that would now seem unJapanese, and products in America will be designed with an insistence on [taste](#) that would now seem unAmerican. Perhaps the most successful countries, in the future, will be those most willing to ignore what are now considered national characters, and do each kind of work in the way that works best. Race you.

多くの技術では、バージョン 2 はより高い目標を達成している。デザイン一般で同じことができるんじゃないか。おそらく、国ごとの性質よりも、職業の持つ性質のほうが次第に勝るようになってゆくだろうと私は思う。日本のハッカーは、今は日本的でないと思われるような、不服従的な振る舞いを表に出すことをゆるされるようになってゆくだろう。米国の製品は、今は米国的でないと思われるような、センスへのこだわりを持ってデザインされるようになってゆくだろう。きっと将来最も成功する国は、今日「お国柄」と思われているようなものを捨てて、それぞれの事柄についてそれが最も有効に働く方法を採っていく国であるだろう。さあ、競争だ。

Notes

注釈

[1] Japanese cities are ugly too, but for different reasons. Japan is prone to earthquakes, so buildings are traditionally seen as temporary; there is no grand tradition of city planning like the one Europeans inherited from Rome. The other cause is the notoriously corrupt relationship between the government and construction companies.

[1] 日本の都市も醜いが、それは別の理由による。地震の多さから、多くの建造物は一時的なものとみなされる傾向があった。またヨーロッパ人がローマから継承した大きな都市計画というものもなかった。他の原因として、政府と建設会社の癒着もある。

Thanks to Trevor Blackwell, Barry Eisler, Sarah Harlin, Shiro Kawai, Jessica Livingston, Jackie McDonough, Robert Morris, and Eric Raymond for reading drafts of this.

Chapter 1 WHY NERDS ARE UNPOPULAR

第1章 どうしてオタクはもてないか

February 2003



When we were in junior high school, my friend Rich and I made a map of the school lunch tables according to popularity. This was easy to do, because kids only ate lunch with others of about the same popularity. We graded them from A to E. A tables were full of football players and cheerleaders and so on. E tables contained the kids with mild cases of Down's Syndrome, what in the language of the time we called "retards."

中学生の時、友だちのリッチと一緒に学校のランチテーブルを人気の度合で区分けした地図を作ったことがある。これは簡単だった。子供は同じくらいの人気度の友達としか固まらないからだ。地図には A から E までのランクを付けた。A テーブルはフットボールの選手とチアリーダーでいっぱい。E テーブルには、軽度のダウン症、当時は「知恵遅れ」と言われていた子たちが座っていた。

We sat at a D table, as low as you could get without looking physically different. We were not being especially candid to grade ourselves as D. It would have taken a deliberate lie to say otherwise. Everyone in the school knew exactly how popular everyone else was, including us.

私たちは D テーブルだった。見かけが普通の子供のなかでは最低のランクだ。私たちがとりわけ率直だったというわけじゃない。D ランク以外にしてたら嘘をついたことになっていただろう。学校の誰もが、私たちも含め、誰がどのくらい人気があるかを正確に知っていた。

My stock gradually rose during high school. Puberty finally arrived; I became a decent soccer player; I started a scandalous underground newspaper. So I've seen a good part of the popularity landscape.

私の人気は徐々に上がり、最終的には、かなり優れたサッカープレーヤーになったし、スキャンダルを扱う学校新聞も書き始めた。だから、私は人気ランクの良い面を見ることができたわけだ。

I know a lot of people who were nerds in school, and they all tell the same story: there is a strong correlation between being smart and being a nerd, and an even stronger inverse correlation between being a nerd and being popular. Being smart seems to *make* you unpopular.

私の知り合いには学校時代にオタクだった人がたくさんいるが、皆同じ話をする。賢いこととオタクであることの間には極めて強い相関があり、オタクであることと人気者であることの間にはもっと強い負の相関があるというものだ。賢いとむしろ不人気者になるみたいなんだ。

Why? To someone in school now, that may seem an odd question to ask. The mere fact is so overwhelming that it may seem strange to imagine that it could be any other way. But it could. Being smart doesn't make you an outcast in elementary school. Nor does it harm you in the real world. Nor, as far as I can tell, is the problem so bad in most other countries. But in a typical American secondary school, being smart is likely to make your life difficult. Why?

どうしてだろう。今学校に通っている子供なら、この質問は変に思えるだろう。だってそれは目の前にある事実であって、そうではない可能性を考えることのほうがおかしいからだ。でも必ずしもそうである必要はないんじゃないか。小学校では賢いからって仲間外れにされることはない。実社会に出てからもそうだ。それに私の知る限り、他の国ではこんなに悪い状況ではないようだ。米国の典型的な中学校に限っては、賢いことは人生を困難にするんだ。なぜだろう。

The key to this mystery is to rephrase the question slightly. Why don't smart kids make themselves popular? If they're so smart, why don't they figure out how popularity works and beat the system, just as they do for standardized tests?

この謎を解く鍵は、質問をちょっとだけ言い換えてみることだ。どうして賢い子供は自分を人気者にしないんだろう。そんなに賢いのなら、人気者になれる方法を見つけ出して逆転すればいいじゃないか。共通テストでやっているように。

One argument says that this would be impossible, that the smart kids are unpopular because the other kids envy them for being smart, and nothing they could do could make them popular. I wish. If the other kids in junior high school envied me, they did a great job of concealing it. And in any case, if being smart

were really an enviable quality, the girls would have broken ranks. The guys that guys envy, girls like.

それは無理だ、賢い子供は他の子供から妬まれるから、という意見がある。何をしようと人気者にはなれないよと。それが真実だったらなあ。中学校で他の生徒が私を妬んでいたんだとしたら、皆はそれをものすごくうまく隠し通していたことになる。それに賢いことが本当に妬みの対象になるんだとしたら、女の子たちはランクづけをぶち壊していたはずだ。男子に妬まれる男子は、女子にもてるはずだろう。

In the schools I went to, being smart just didn't matter much. Kids didn't admire it or despise it. All other things being equal, they would have preferred to be on the smart side of average rather than the dumb side, but intelligence counted far less than, say, physical appearance, charisma, or athletic ability.

私が通っていた学校では賢いことは単に重視されなかった。称賛もされなかったし、嫌われることもなかった。他の全てが同じなら平均より賢いほうがその逆よりは好まれただろう。それでも頭の良さは他のもの、外見とか魅力とか運動能力に比べたら、ずっと軽視されていた。

So if intelligence in itself is not a factor in popularity, why are smart kids so consistently unpopular? The answer, I think, is that they don't really want to be popular.

知性そのものが人気を決める要素でないとしたら、どうして賢い子供はいつも不人気なんだろうか。私が考える答えは、そういう子供は本当は人気者になりたいとは思っていないということだ。

If someone had told me that at the time, I would have laughed at him. Being unpopular in school makes kids miserable, some of them so miserable that they commit suicide. Telling me that I didn't want to be popular would have seemed like telling someone dying of thirst in a desert that he didn't want a glass of water. Of course I wanted to be popular.

いや、当時の私に誰かがそんなふうにしたら、笑っちゃっていたと思うよ。学校で人気がないってことは灰色の学生生活を意味する。あんまり辛いんで自殺を試みる子さえいるくらいだ。本当は人気者になりたいんじゃないか、なんて言うことは、砂漠の真ん中で渇きに苦しむ人に水なんか欲しくないだろうと言うようなものだ。そりゃあ人気者になりたかったさ。

But in fact I didn't, not enough. There was something else I wanted more: to be

smart. Not simply to do well in school, though that counted for something, but to design beautiful rockets, or to write well, or to understand how to program computers. In general, to make great things.

でも実際にはそうでなかった。それほどには。私にはそれよりもっとなりたいたことがあったんだ。賢くなることだ。学校で良い成績を収めることだけじゃない。それにも意味はあったけれど、それより美しいロケットを設計したいとか、良い文章を書きたいとか、コンピュータをプログラムする方法を理解したいとか、とにかくすごいものを作りたかったんだ。

At the time I never tried to separate my wants and weigh them against one another. If I had, I would have seen that being smart was more important. If someone had offered me the chance to be the most popular kid in school, but only at the price of being of average intelligence (humor me here), I wouldn't have taken it.

当時の私は、自分の欲求を整理して比べてみたりはしなかった。それをしていれば、自分にとっては賢くなるほうが大事なんだと気付いていただろう。誰かに、一番の人気者にしてやる、その代わりに知能指数は並(笑)で我慢しろ、と言われたとしても、その取り引きには応じなかつただろうと思う。

Much as they suffer from their unpopularity, I don't think many nerds would. To them the thought of average intelligence is unbearable. But most kids would take that deal. For half of them, it would be a step up. Even for someone in the eightieth percentile (assuming, as everyone seemed to then, that intelligence is a scalar), who wouldn't drop thirty points in exchange for being loved and admired by everyone?

人気がないことでいくら苦しんでいても、多くのオタクはその取り引きに応じないはずだ。並の頭になることはオタクにとっては耐え難いことだからだ。でも普通の子供なら応じるだろうと思う。半分の子供たちにとってはステップアップになるわけだし。(当時皆が思っていたように、知性がスカラー値だったとして)80 ポイントの位置にいる子供だって、誰からも愛され、もてはやされるなら、30 ポイント分と交換してもいいと思うんじゃないだろうか。

And that, I think, is the root of the problem. Nerds serve two masters. They want to be popular, certainly, but they want even more to be smart. And popularity is not something you can do in your spare time, not in the fiercely competitive environment of an American secondary school.

それこそが、私が思うに、問題の根っ子なんだ。オタクは二君に仕えてるんだ。人気者にはもちろんなりたいが、賢くなりたいという望みのほうがずっと強い。そして、人気を得ることは余暇に何とかできるようなものじゃない。特に米国の中学校のような、凶暴なまでに競争の激しい環境では。

Alberti, arguably the archetype of the Renaissance Man, writes that "no art, however minor, demands less than total dedication if you want to excel in it." I wonder if anyone in the world works harder at anything than American school kids work at popularity. Navy SEALs and neurosurgery residents seem slackers by comparison. They occasionally take vacations; some even have hobbies. An American teenager may work at being popular every waking hour, 365 days a year.

ルネッサンス初期を代表する天才アルベルティは、「人生の全てを賭けずして極められる技などない」と書いている。米国の学校の生徒が人気を得るために努力するほどに、熱心に頑張っている人間は世界中探しても他にいないのではないか。海軍の特殊部隊や神経外科医のインターンだってそれに比べればゆったりしたものだ。休暇を取ったり趣味を持つことさえできるからね。米国のティーンは人気を得るために、起きている時間全て、365日絶え間なく頑張っている。

I don't mean to suggest they do this consciously. Some of them truly are little Machiavellis, but what I really mean here is that teenagers are always on duty as conformists.

意識してやっているわけじゃないだろう。子供たちの一部は本当に小さなマキャヴェリと言えるだろうけれど、ここで言いたいのは、ティーンエイジャーは四六時中規律に従う遵奉者であるということだ。

For example, teenage kids pay a great deal of attention to clothes. They don't consciously dress to be popular. They dress to look good. But to who? To the other kids. Other kids' opinions become their definition of right, not just for clothes, but for almost everything they do, right down to the way they walk. And so every effort they make to do things "right" is also, consciously or not, an effort to be more popular.

例えばティーンの子供は服装にものすごく気を遣う。意識して人気者になるために服を選んでるわけじゃないだろう。ただ格好良くしていただけたらいい。でもそれは誰にとってだ?他

の子供たちにだろう。他の子供たちの意見が、服装だけでなく歩き方から何から全てについて、何が正しいかを規定する。だから、何でも「正しく」やろうとする努力は、意識するにせよしないにせよ、人気を得ようとする努力にもなるんだ。

Nerds don't realize this. They don't realize that it takes work to be popular. In general, people outside some very demanding field don't realize the extent to which success depends on constant (though often unconscious) effort. For example, most people seem to consider the ability to draw as some kind of innate quality, like being tall. In fact, most people who "can draw" like drawing, and have spent many hours doing it; that's why they're good at it. Likewise, popular isn't just something you are or you aren't, but something you make yourself.

オタクにはこれが分からない。人気者になるには努力が必要だということに気付かないんだ。一般的に、大変な努力が必要な分野を外から見ている人には、成功するためにどれだけの絶え間ない頑張りが(しばしば無意識のうちに)必要かということが見えにくい。例えば絵を描く能力は、ちょうど背の高さみたいに天賦の資質だと思っている人は多いだろう。実際は「絵が描ける」人は描くことが好きで多くの時間をそれに費やしたんだ。だからうまく描けるんだ。同じように、人気者ということも天賦の資質ではない。人は努力して人気者になるんだ。

The main reason nerds are unpopular is that they have other things to think about. Their attention is drawn to books or the natural world, not fashions and parties. They're like someone trying to play soccer while balancing a glass of water on his head. Other players who can focus their whole attention on the game beat them effortlessly, and wonder why they seem so incapable.

オタクがもてない主要な理由は、他に考えることがあるからだ。オタクの注意力はどうしても、ファッションやパーティよりは本や自然に向けられてしまう。これじゃ頭に水の入ったコップを載せてサッカーをしようとしているみたいなものだ。他のプレイヤーはゲームだけに集中して、そんなオタクを簡単に負かしてしまうだろう。そしてどうしてオタクたちはこんなに下手くそなんだろうと頭をひねるんだ。

Even if nerds cared as much as other kids about popularity, being popular would be more work for them. The popular kids learned to be popular, and to want to be popular, the same way the nerds learned to be smart, and to want to be smart: from their parents. While the nerds were being trained to get the right answers, the popular kids were being trained to please.

たとえオタクが他の子供たちと同じくらい人気者になろうと頑張ったとしても、それは他

の子供たちよりも大変だろう。オタクは賢くなる方法を教わり、人気者になりたいと望むことを教わる。人気のある子供は人気を得る方法を教わり、人気者になりたいと望むことを教わる。自分の親たちから。オタクは正しい答えを返すようにしつけられてきたが、人気のある子供は親を喜ばせるようにしつけられてきたんだ。

So far I've been finessing the relationship between smart and nerd, using them as if they were interchangeable. In fact it's only the context that makes them so. A nerd is someone who isn't socially adept enough. But "enough" depends on where you are. In a typical American school, standards for coolness are so high (or at least, so specific) that you don't have to be especially awkward to look awkward by comparison.

ここまでは、賢いこととオタクであることとの関係を、それらが交換可能であるかのように扱ってきた。でもそうになっているのは環境のせいにすぎない。オタクとは社会性において十分な能力がない者のことをいう。だが「十分な」といっても、その大きさは、あなたがどこにいるかによって変わってくる。典型的な米国の学校では格好良さの基準があまりに高く(少なくとも、あまりに特殊で)、そのせいで特に不格好でなくても比較されると不格好に見えてしまうんだ。

Few smart kids can spare the attention that popularity requires. Unless they also happen to be good-looking, natural athletes, or siblings of popular kids, they'll tend to become nerds. And that's why smart people's lives are worst between, say, the ages of eleven and seventeen. Life at that age revolves far more around popularity than before or after.

人気者になるには注意力が必要だ。そして、賢い子供には、そんなことに回せるムダな注意力など残っていないのだ。たまたまルックスが良いとか、生まれついて運動能力が高いとか、人気者の兄弟だとかいうのでもなければ、オタクになるしかない。

Before that, kids' lives are dominated by their parents, not by other kids. Kids do care what their peers think in elementary school, but this isn't their whole life, as it later becomes.

それより下の年代の子供の生活は、他の子供よりも親によって支配されている。小学生でも、もちろん他の子供がどう考えるか気にするけれど、それは後にそうなるように、人生の全ての問題というわけじゃない。

Around the age of eleven, though, kids seem to start treating their family as a day job. They create a new world among themselves, and standing in this world is what matters, not standing in their family. Indeed, being in trouble in their family can win them points in the world they care about.

しかし 11 歳前後になると、子供は仕事で仕方なく付き合うみたいに親と接するようになる。仲間内だけの世界を作り始める。そして、家族の中の立場ではなく、その世界での立場が重要になる。むしろ家族と喧嘩したことは、もう一方のもっと大事な世界でプラス点になるくらいだ。

The problem is, the world these kids create for themselves is at first a very crude one. If you leave a bunch of eleven-year-olds to their own devices, what you get is *Lord of the Flies*. Like a lot of American kids, I read this book in school. Presumably it was not a coincidence. Presumably someone wanted to point out to us that we were savages, and that we had made ourselves a cruel and stupid world. This was too subtle for me. While the book seemed entirely believable, I didn't get the additional message. I wish they had just told us outright that we were savages and our world was stupid.

問題は、子供が自分たちで作る世界は最初はとても残酷な世界だということだ。11 歳の子供たちを好きにさせておいたら、そこにできるのは『蠅の王』の世界だろう。他の多くの米国人の子供と同じように、私もこの本を学校で読まされた。もしかするとこれは偶然じゃなかったんじゃないか。私たちは野蛮人で、自ら残酷で愚かな世界を作り出したんだと誰かが知らせたかったんじゃないか。この本は私には難しすぎた。確かにありそうな話だとは思ったが、それ以上のメッセージを得ることはできなかった。僕らは野蛮人で、僕らの作った世界は馬鹿げている、そう誰かがはっきりと教えてくれたらよかったのに。

Nerds would find their unpopularity more bearable if it merely caused them to be ignored. Unfortunately, to be unpopular in school is to be actively persecuted.

不人気であることが単に無視されるだけなら、オタクにとってもまだ耐えやすいことだったろう。残念ながら、学校で不人気であるということは、積極的にいじめられるということなんだ。

Why? Once again, anyone currently in school might think this a strange question to ask. How could things be any other way? But they could be. Adults don't normally persecute nerds. Why do teenage kids do it?

なぜだろう。このことも、今学校にいる子供には変な質問だと思えるだろう。そうじゃないことなんてあり得るのかってね。あり得るさ。普通の大人はオタクをいじめたりしない。どうしてティーンの子供はそうするんだろう。

Partly because teenagers are still half children, and many children are just intrinsically cruel. Some torture nerds for the same reason they pull the legs off spiders. Before you develop a conscience, torture is amusing.

理由のひとつは、ティーンエイジャーはまだ半分子供で、多くの子供は本質的に残酷なものだからだ。クモの足を引きちぎって遊ぶのと同じ理由からオタクをいじめる子供だっている。良心を培う以前には拷問は楽しい。

Another reason kids persecute nerds is to make themselves feel better. When you tread water, you lift yourself up by pushing water down. Likewise, in any social hierarchy, people unsure of their own position will try to emphasize it by maltreating those they think rank below. I've read that this is why poor whites in the United States are the group most hostile to blacks.

子供がオタクを迫害するもうひとつの理由は、そうすることで優越感に浸れるからだ。立ち泳ぎする時は、水を押し下げることで浮き上がることができる。どのように、どんな社会階級においても、自分の立場が不安な人々は、自分より下の階級だと思う人々を手ひどく扱うことで足場を固めようとする。米国で黒人に最も強い敵意を示すのは貧しい白人階級であることの理由はそれだ、という説を読んだことがある。

But I think the main reason other kids persecute nerds is that it's part of the mechanism of popularity. Popularity is only partially about individual attractiveness. It's much more about alliances. To become more popular, you need to be constantly doing things that bring you close to other popular people, and nothing brings people closer than a common enemy.

だが私が思うに、他の子供がオタクを迫害する一番の理由は、それが人気のメカニズムの一部だからだ。個人の魅力というのは人気の一部にすぎない。むしろ同盟関係が重要なんだ。より人気を得るには他の人気者の側にいられるようなことをやり続けなければならない。共通の敵を持つことは、人が結束する最良の方法だ。

Like a politician who wants to distract voters from bad times at home, you can create an enemy if there isn't a real one. By singling out and persecuting a nerd, a group of kids from higher in the hierarchy create bonds between themselves. Attacking an outsider makes them all insiders. This is why the worst cases of

bullying happen with groups. Ask any nerd: you get much worse treatment from a group of kids than from any individual bully, however sadistic.

支持率が低迷している時に政治家が有権者の気をそらせたがるように、本当の敵がいなくても敵をでっち上げることができる。オタクをあぶり出して迫害することで、上の階級にいる子供たちはより強く結束する。よそ者を攻撃すればみな仲間になれる。いじめの最悪の形態が集団の中で生まれるのはそのせいだ。どんなオタクに聞いても、個人にされるいじめはいかにひどくても、集団でやられるよりはましだと言うだろう。

If it's any consolation to the nerds, it's nothing personal. The group of kids who band together to pick on you are doing the same thing, and for the same reason, as a bunch of guys who get together to go hunting. They don't actually hate you. They just need something to chase.

オタクにとって慰めになるとすれば、それは人格とは何の関係もないものだってことだ。一緒につるんでオタクをいじめる子供たちは、男たちがつるんで狩りに行くのと同じことを、同じ理由でやっているんだ。オタクを本当に嫌っているわけじゃない。ただ何か追いかけるものが必要なだけなんだ。

Because they're at the bottom of the scale, nerds are a safe target for the entire school. If I remember correctly, the most popular kids don't persecute nerds; they don't need to stoop to such things. Most of the persecution comes from kids lower down, the nervous middle classes.

オタクが一番下の階級にいるから、学校の中で一番安全な標的だ。私の記憶が正しければ、一番の人気者はオタクを迫害することはなかった。そんなことをする必要がないからだ。ほとんどのいじめはもっと下のほう、不安を感じている中流ランクの子供たちが火付け役になっている。

The trouble is, there are a lot of them. The distribution of popularity is not a pyramid, but tapers at the bottom like a pear. The least popular group is quite small. (I believe we were the only D table in our cafeteria map.) So there are more people who want to pick on nerds than there are nerds.

困ったことにそういう中流ランクの子供たちはたくさんいる。人口分布はピラミッドではなく、洋ナシのように底のほうが細くなっているんだ。一番人気のないグループはとても少ない(D グループは私たちのテーブルだけだったと思う)。だからオタクよりもオタクをいじめたいと思う子供のほうがずっと多いんだ。

As well as gaining points by distancing oneself from unpopular kids, one loses

points by being close to them. A woman I know says that in high school she liked nerds, but was afraid to be seen talking to them because the other girls would make fun of her. Unpopularity is a communicable disease; kids too nice to pick on nerds will still ostracize them in self-defense.

人気のない子供たちから離れることでポイントを得られるのと同様に、彼らに近づくことでポイントを失ってしまう。私の知人のある女性は、オタクは好きだったけど、オタクと話しているところを見られたら他の女の子に馬鹿にされると思っていた、と高校時代のことを話していた。嫌われ者は伝染病だ。オタクをいじめるには善良過ぎる子供でも、自己防衛のためにオタクを仲間外れにするんだ。

It's no wonder, then, that smart kids tend to be unhappy in middle school and high school. Their other interests leave them little attention to spare for popularity, and since popularity resembles a zero-sum game, this in turn makes them targets for the whole school. And the strange thing is, this nightmare scenario happens without any conscious malice, merely because of the shape of the situation.

これじゃあ中学高校で賢い子供が不幸な生活を送るのは無理もない。他に興味の対象があるせいで、人気を得るために使うエネルギーなどほとんどない。しかも人気というのはゼロサムゲームに似ているために、結果として他の皆の標的になってしまうんだ。奇妙なのは、この悪夢は意識的な悪意から生み出されるわけじゃなく、単なる状況設定によって生ずるものだということだ。

For me the worst stretch was junior high, when kid culture was new and harsh, and the specialization that would later gradually separate the smarter kids had barely begun. Nearly everyone I've talked to agrees: the nadir is somewhere between eleven and fourteen.

私にとって一番辛かったのは中学時代だった。未成熟の子供の文化は残酷で、優秀な子供たちが自分の好きな分野で能力を伸ばせる機会などほとんどなかった。この話をすると、ほとんどの人が同意してくれる。最悪なのは 11 歳から 14 歳の間だったと。

In our school it was eighth grade, which was ages twelve and thirteen for me. There was a brief sensation that year when one of our teachers overheard a group of girls waiting for the school bus, and was so shocked that the next day she devoted the whole class to an eloquent plea not to be so cruel to one another.

第 8 学年のころ、12 歳から 13 歳になるところだったが、ちょっとした事件があった。ス

クールバスを待つ女子たちの会話をある先生が耳にして大きなショックを受け、翌日の授業を丸々潰して、お互いにもっと思いやりを持つように切々と訴えたんだ。

It didn't have any noticeable effect. What struck me at the time was that she was surprised. You mean she doesn't know the kind of things they say to one another? You mean this isn't normal?

目に見えた変化は何も起こらなかった。私は、先生が驚いたということ自体に驚いたものだ。女子がどんな会話をしているか知らなかったとでもいうのかい?そして、彼女らの会話が普通じゃないとでも?

It's important to realize that, no, the adults don't know what the kids are doing to one another. They know, in the abstract, that kids are monstrously cruel to one another, just as we know in the abstract that people get tortured in poorer countries. But, like us, they don't like to dwell on this depressing fact, and they don't see evidence of specific abuses unless they go looking for it.

大人は子供がお互いに何をしているかを分かっていない。そして、その分かっていないということに気付くべきだ。子供はぞっとするほど残酷な仕打ちを相手にするものだが、そのことを大人はうっすらとしか知らない。これは私たちが、貧しい国々でまだ拷問が行われているということをぼんやりとしか知らないのと同じだ。大人は気が滅入る事実に対して正面から向き合おうとはしない。具体的ないじめの証拠は、意識して探さない限り目に入らないんだ。

Public school teachers are in much the same position as prison wardens. Wardens' main concern is to keep the prisoners on the premises. They also need to keep them fed, and as far as possible prevent them from killing one another. Beyond that, they want to have as little to do with the prisoners as possible, so they leave them to create whatever social organization they want. From what I've read, the society that the prisoners create is warped, savage, and pervasive, and it is no fun to be at the bottom of it.

公立高校の教師は、刑務所の看守と似たような立場にある。看守の役目はとにかく囚人を閉じ込めておくことだ。食事を取らせなければならないし、喧嘩で死人が出るなんてことのないように目を光らせている必要もある。でも必要以上に囚人とかわりたくないから、囚人が好きなように社会を作るのを放置している。私が読んで知る限りでは、それは常軌を逸した残酷な倒錯の世界だ。その底辺にいる人間は、心が休まることはないだろう。

In outline, it was the same at the schools I went to. The most important thing was

to stay on the premises. While there, the authorities fed you, prevented overt violence, and made some effort to teach you something. But beyond that they didn't want to have too much to do with the kids. Like prison wardens, the teachers mostly left us to ourselves. And, like prisoners, the culture we created was barbaric.

私が通っていた学校でも状況は似たようなものだった。一番大事なことは校内にとどまっていることだった。そうしていれば食事が与えられ、あからさまな暴力から守られ、何かを教えてもらえる。でも学校がしてくれるのはそこまでだ。看守と同じように教師は子供に好き勝手をやらせておく。そして囚人と同じように生徒は残酷な社会を作り出すのだ。

Why is the real world more hospitable to nerds? It might seem that the answer is simply that it's populated by adults, who are too mature to pick on one another. But I don't think this is true. Adults in prison certainly pick on one another. And so, apparently, do society wives; in some parts of Manhattan, life for women sounds like a continuation of high school, with all the same petty intrigues.

どうして実社会のほうがオタクに優しいんだろう。いじめなんか卒業した大人によって構成されているから、という簡単な理由だろうか。そんなことはない。刑務所では大人の間だっていじめがある。社交界の御婦人方だってそうだ。マンハッタンの一部では、女性は大人になってからも、高校時代に受けていたような嫌がらせに耐え続けなければならない。

I think the important thing about the real world is not that it's populated by adults, but that it's very large, and the things you do have real effects. That's what school, prison, and ladies-who-lunch all lack. The inhabitants of all those worlds are trapped in little bubbles where nothing they do can have more than a local effect. Naturally these societies degenerate into savagery. They have no function for their form to follow.

実社会は大人の集まりだからいいというわけではない。大切なのは、行動規範が広いということ、そして自らの行動を通じて環境を変えることができるということだ。学校や刑務所、御夫人方の昼食会にはこれが欠けている。このような世界の住人は小さな泡の中に捕らわれていて、自分の影響が及ぶのはその泡の中だけだ。このような社会は野蛮なものへと退化してゆくしかない。改善につながる圧力がどこからもかからないのだから。

When the things you do have real effects, it's no longer enough just to be pleasing. It starts to be important to get the right answers, and that's where nerds show to

advantage. Bill Gates will of course come to mind. Though notoriously lacking in social skills, he gets the right answers, at least as measured in revenue.

自らの行動が回りに影響を及ぼす世界では、ただ楽しめればいいでは済まされない。正しい答えを得ることが重要になる。そこで力を発揮するのがオタクで、その代表格がビル・ゲイツだ。彼は社交性が決定的に欠けているにもかかわらず、少なくとも収益で測った場合には、正しい答えを得ている。

The other thing that's different about the real world is that it's much larger. In a large enough pool, even the smallest minorities can achieve a critical mass if they clump together. Out in the real world, nerds collect in certain places and form their own societies where intelligence is the most important thing. Sometimes the current even starts to flow in the other direction: sometimes, particularly in university math and science departments, nerds deliberately exaggerate their awkwardness in order to seem smarter. John Nash so admired Norbert Wiener that he adopted his habit of touching the wall as he walked down a corridor.

実社会でもうひとつ異なるのはその大きさだ。母集団が十分に大きいと、最も小さいマイノリティでも結束すれば影響力を持つに十分な閾値を超えられる。実社会ではオタクたちが集まって、知性こそが最重要視される自分たちの世界を作る。時には流れが逆を向くことだってある。特に大学の理学部では、オタクは自分を賢く見せるためにわざと不器用さを強調することさえあるんだ。ジョン・ナッシュはノーバート・ウィーナーを崇拝するあまり、廊下を歩くときに壁を触るという癖を真似していた。

As a thirteen-year-old kid, I didn't have much more experience of the world than what I saw immediately around me. The warped little world we lived in was, I thought, *the world*. The world seemed cruel and boring, and I'm not sure which was worse.

13歳の子供のころ、私は自分のすぐ周りにある世界の外側のことは何も知らなかった。自分の住んでいた小さな歪んだ世界が全てだった。世界は残酷でしかも退屈で、自分にとってどっちが悪いのかもよく分からなかった。

Because I didn't fit into this world, I thought that something must be wrong with me. I didn't realize that the reason we nerds didn't fit in was that in some ways we were a step ahead. We were already thinking about the kind of things that matter in the real world, instead of spending all our time playing an exacting but mostly

pointless game like the others.

この世界に馴染めないのは、私にどこか悪いところがあるからだと思っていた。オタクが世界に馴染めないのは、ある意味一歩先を行っていたからだということに気付かなかったんだ。オタクたちは既に、他の子供たちとは違って、骨の折れるだけの無意味なゲームに全ての時間を費やすなんてことはせず、実社会で問題になるような事柄を考え始めていたんだ。

We were a bit like an adult would be if he were thrust back into middle school. He wouldn't know the right clothes to wear, the right music to like, the right slang to use. He'd seem to the kids a complete alien. The thing is, he'd know enough not to care what they thought. We had no such confidence.

ある意味、私たちは中学校に呼び戻された大人のようなものだった。中学生がどういう服装をして、どういう音楽を聞いて、どういうスラングを使ったらさまになるかなんて全然わからない。きっと全く別種の人間のように見えたはずだ。ただ、もしそうだったとしたら、同級生にどう思われるかなんて気にしなくてもいいくらいものを知っていたことだろう。当時の私たちにはその自信がなかった。

A lot of people seem to think it's good for smart kids to be thrown together with "normal" kids at this stage of their lives. Perhaps. But in at least some cases the reason the nerds don't fit in really is that everyone else is crazy. I remember sitting in the audience at a "pep rally" at my high school, watching as the cheerleaders threw an effigy of an opposing player into the audience to be torn to pieces. I felt like an explorer witnessing some bizarre tribal ritual.

世間では、賢い子供たちがこの年代に「普通の」子供たちの中で一緒に過ごすことが良いことだと信じられているようだ。そうかもしれない。でも、他の皆がいかれているせいで、オタクが浮いて見えるというケースだってあるにはある。高校の頃の話だが、私は、応援団の「壮行会」の観客席にいて、チアリーダーが敵のプレーヤーに見立てた人形を観客に向けて投げ込み、それがずたずたに引き裂かれるのを見たのを覚えている。何か奇妙な種族の儀式を目撃している探検家になった気分だった。

If I could go back and give my thirteen year old self some advice, the main thing I'd tell him would be to stick his head up and look around. I didn't really grasp it at the time, but the whole world we lived in was as fake as a Twinkie. Not just school, but the entire town. Why do people move to suburbia? To have kids! So no

wonder it seemed boring and sterile. The whole place was a giant nursery, an artificial town created explicitly for the purpose of breeding children.

時を遡って 13 歳の自分にアドバイスできるとしたら、まず頭を上げて周りを見てみると言
ってやりたい。当時は気付いていなかったのだが、自分たちがいた世界の全ては張りぼて
の偽物だった。学校だけじゃない。町の全てがだ。人々が郊外に引っ越す理由は何だい？子
供を作るためだろう。そこが退屈で空虚な場所になるのも当然だ。それ全体が、巨大な育
児室みたいなものだからね。子供を育てるためだけに作られた人工の町だ。

Where I grew up, it felt as if there was nowhere to go, and nothing to do. This was
no accident. Suburbs are deliberately designed to exclude the outside world,
because it contains things that could endanger children.

私が育った町では、他にどこにも行くところがなく、何もすることがないように感じられ
た。これも当然といえるだろう。そもそも郊外は、子供にとっての危険が潜んでいる外界
とは意図的に切り離されて開発されたものだ。

And as for the schools, they were just holding pens within this fake world.
Officially the purpose of schools is to teach kids. In fact their primary purpose is to
keep kids locked up in one place for a big chunk of the day so adults can get things
done. And I have no problem with this: in a specialized industrial society, it would
be a disaster to have kids running around loose.

そして学校も同様に、この偽りの世界の一端を担っている。学校の表向きの目的は子供を
教育することだ。しかし、本当の目的は、日中、子供を一箇所に閉じ込めておいて、大人
が仕事ができるようにすることにある。これについては私は問題にしない。実際、専門化
された工業社会で子供を勝手に遊ばせておくと、とんでもないことになる。

What bothers me is not that the kids are kept in prisons, but that (a) they aren't
told about it, and (b) the prisons are run mostly by the inmates. Kids are sent off
to spend six years memorizing meaningless facts in a world ruled by a caste of
giants who run after an oblong brown ball, as if this were the most natural thing
in the world. And if they balk at this surreal cocktail, they're called misfits.

問題なのは、子供が刑務所に囚われていることではなく、(a)そのことを子供が知らされて
いないこと、そして(b)その刑務所が主として収容者自身によって運営されていることだ。
まるでそれが世界で一番自然なことであるように、茶色い楕円のボールを追いかけている
特権階級の大男たちによって支配されている世界で、子供たちは無意味な事実を暗記する
ために 6 年を費やすんだ。そしてこの非現実的なごちゃごちゃの世界に文句を付けようも
のなら、不適応者と呼ばれてしまう。

Life in this twisted world is stressful for the kids. And not just for the nerds. Like any war, it's damaging even to the winners.

このように歪んだ世界は子供にとって大きな負担だ。オタクにとってだけじゃない。他のどんな戦争とも同じように、勝者さえも傷つけられる。

Adults can't avoid seeing that teenage kids are tormented. So why don't they do something about it? Because they blame it on puberty. The reason kids are so unhappy, adults tell themselves, is that monstrous new chemicals, *hormones*, are now coursing through their bloodstream and messing up everything. There's nothing wrong with the system; it's just inevitable that kids will be miserable at that age.

大人だって、ティーンエイジの子供たちが苦しんでいるのが見えないわけがない。それなのにどうして何もしないかって？思春期のせいになっているからだ。子供たちがこんなにふこなのは、恐ろしい科学物質、ホルモンってやつが、血管の中をかけめぐって何もかもをめちゃくちゃにしているからだ、と大人は自分に言い聞かせている。制度には何も問題がない。子供がああ年齢の時に苦しむのは避けられないんだ、と。

This idea is so pervasive that even the kids believe it, which probably doesn't help. Someone who thinks his feet naturally hurt is not going to stop to consider the possibility that he is wearing the wrong size shoes.

この考え方は完璧に浸透していて当の子供たちでさえ信じているくらいだが、何の役にも立たない。常日頃、足に痛みを感じている人が、靴のサイズが合っていないという可能性を考えないなんてことがあるだろうか。

I'm suspicious of this theory that thirteen-year-old kids are intrinsically messed up. If it's physiological, it should be universal. Are Mongol nomads all nihilists at thirteen? I've read a lot of history, and I have not seen a single reference to this supposedly universal fact before the twentieth century. Teenage apprentices in the Renaissance seem to have been cheerful and eager. They got in fights and played tricks on one another of course (Michelangelo had his nose broken by a bully), but they weren't crazy.

この、13歳の子供がめちゃくちゃなのはどうしようもないという理論は、かなり怪しいと私は思っている。それが生理的なものなら普遍的に見られるはずだろう。モンゴルの遊牧

民は 13 歳の時にニヒリストになるだろうか。歴史書をたくさん読んだが、20 世紀以前にこの普遍的であるはずの理論に言及されたものを見たことがない。ルネッサンス時代のティーンエイジの見習いたちは、陽気で熱心であったようだ。もちろん喧嘩したりお互いに悪戯をしたりということはあっただろう(ミケランジェロは、いじめっ子に鼻の骨を折られたことがある)。でも彼らはいかれてはいなかった。

As far as I can tell, the concept of the hormone-crazed teenager is coeval with suburbia. I don't think this is a coincidence. I think teenagers are driven crazy by the life they're made to lead. Teenage apprentices in the Renaissance were working dogs. Teenagers now are neurotic lapdogs. Their craziness is the craziness of the idle everywhere.

私の知る限り、ホルモンでティーンエイジャーがおかしくなるという考え方は、郊外の街の発生と時を同じくしている。これは偶然ではないと思う。ティーンの子供は、選択の余地なく送り込まれる生活そのものによっておかしくなるんだ。ルネッサンスのティーンの丁稚たちは馬車馬のように働いた。現代のティーンたちはノイローゼの座敷犬だ。何もしないでいることからおかしくなってしまうんだ。

When I was in school, suicide was a constant topic among the smarter kids. No one I knew did it, but several planned to, and some may have tried. Mostly this was just a pose. Like other teenagers, we loved the dramatic, and suicide seemed very dramatic. But partly it was because our lives were at times genuinely miserable.

私の学校時代、聡明な子供たちの間では常に自殺が話題に上っていた。私が知るなかで実際に実行した人はいないが、何人かは計画していたし、未遂に終わった人もいたかもしれない。たいていは単なるポーズだった。他のティーンと同じように私たちもドラマチックなことが好きだったし、自殺というのはとてもドラマチックなことのように思えたんだ。でももうひとつの理由は、毎日があまりに根本から悲惨だったからだ。

Bullying was only part of the problem. Another problem, and possibly an even worse one, was that we never had anything real to work on. Humans like to work; in most of the world, your work is your identity. And all the work we did was [pointless](#), or seemed so at the time.

いじめは問題の一部にすぎない。もうひとつの、おそらくもっと深刻な問題は、私たちには打ち込めるものが何もなかったということだ。人間は働きたがるものだ。世界の多くの

国々では、仕事がアイデンティティでさえある。でも私たちがやらされていたのは意味のないことだった。少なくとも当時はそう感じられた。

At best it was practice for real work we might do far in the future, so far that we didn't even know at the time what we were practicing for. More often it was just an arbitrary series of hoops to jump through, words without content designed mainly for testability. (The three main causes of the Civil War were.... Test: List the three main causes of the Civil War.)

遠い将来やることになる本物の仕事の練習ってことだったらまだよかったのだろうけど、当時私たちはいったい何のために練習をしているのかさえ分からなかった。無作為に並べられた輪をくぐらされているだけのように感じられたものだ。テストのためだけに用意された、中身のない言葉(南北戦争の 3 つの主要な原因はかくかくしかじか……。テスト：南北戦争の 3 つの主要な理由を挙げよ)。

And there was no way to opt out. The adults had agreed among themselves that this was to be the route to college. The only way to escape this empty life was to submit to it.

そしてそこから抜ける方法はなかった。大人は、それが大学への道だということでは一致しているようだった。この空虚な日々から逃れる唯一の方法は、それに従うことだった。

Teenage kids used to have a more active role in society. In pre-industrial times, they were all apprentices of one sort or another, whether in shops or on farms or even on warships. They weren't left to create their own societies. They were junior members of adult societies.

ティーンエイジの子供たちはかつては社会の中でもっと積極的な役割を担っていた。工業化以前には子供たちはみんな、商人や、農家や、あるいは戦士などの、何かの職業の見習いだった。自分たちの社会を勝手に作れるようにはなっていなかった。大人の社会の準構成員だったんだ。

Teenagers seem to have respected adults more then, because the adults were the visible experts in the skills they were trying to learn. Now most kids have little idea what their parents do in their distant offices, and see no connection (indeed, there is precious little) between schoolwork and the work they'll do as adults.

その当時は、子供たちは大人をもっと尊敬していたようだ。大人は、子供たちが身に付け

ようとしている技能に関して明らかに熟練者だったからだ。現代では、多くの子供は親が遠くのオフィスでいったい何をやっているのかほとんど知らず、したがって学校でやることと将来大人としてやることの間に全く関連を見出せない(もっとも関連なんてほとんどないのだが)。

And if teenagers respected adults more, adults also had more use for teenagers. After a couple years' training, an apprentice could be a real help. Even the newest apprentice could be made to carry messages or sweep the workshop.

ティーンは大人を尊敬し、大人もまたティーンを頼りにしていた。2年ほど修行を積み、見習いは実際の助けになった。新入りの弟子だって、使いをしたり工房を掃除したりといったことくらいはできたんだ。

Now adults have no immediate use for teenagers. They would be in the way in an office. So they drop them off at school on their way to work, much as they might drop the dog off at a kennel if they were going away for the weekend.

現代では大人が子供を直接頼りにするようなことなんてない。仕事場にいれば邪魔になる。だから、仕事に行く途中で学校に子供を預けてゆくんだ。週末に旅行に出る時にペットホテルに犬を預けてゆくみたいに。

What happened? We're up against a hard one here. The cause of this problem is the same as the cause of so many present ills: specialization. As jobs become more specialized, we have to train longer for them. Kids in pre-industrial times started working at about 14 at the latest; kids on farms, where most people lived, began far earlier. Now kids who go to college don't start working full-time till 21 or 22. With some degrees, like MDs and PhDs, you may not finish your training till 30.

何が起きたんだろう。ここからが難しいところだ。この問題の原因は、現代の他の多くの病と同じく、専門化だ。職業が専門化するにつれ、訓練にかかる時間が長くなっていった。工業化以前の社会では、子供は遅くとも14歳までには働き始めていた。ほとんどの人々が生活していた農家ではもっとずっと早くからだ。現代では、大学へ行く子供は21歳か22歳になるまでフルタイムでは働き始めない。医師や博士の学位を得るには、30歳まで見習いを続けなければならないこともあるだろう。

Teenagers now are useless, except as cheap labor in industries like fast food, which evolved to exploit precisely this fact. In almost any other kind of work, they'd be a net loss. But they're also too young to be left unsupervised. Someone

has to watch over them, and the most efficient way to do this is to collect them together in one place. Then a few adults can watch all of them.

現代のティーンは、ファストフードのような安い労働力を使う産業以外では役に立たない。実際、そういう業種はまさにその事実を利用することで拡大したんだ。他のほとんどの職業ではティーンはかえって足手まといになるだろう。でも、まだ監督なしで放っておけるほど成長しておらず、誰かが見ていなければならない。だとしたら、一番効率の良い方法は彼らを一箇所に閉じ込めておくことだ。そうすれば数人の大人が全員をみられる。

If you stop there, what you're describing is literally a prison, albeit a part-time one. The problem is, many schools practically do stop there. The stated purpose of schools is to educate the kids. But there is no external pressure to do this well. And so most schools do such a bad job of teaching that the kids don't really take it seriously-- not even the smart kids. Much of the time we were all, students and teachers both, just going through the motions.

この説明だけを聞けば、誰だって、ああ刑務所の説明をしているんだなと思うだろう。違うのは、時間になったら帰してもらえるということだけだ。そして、多くの学校がまさに刑務所と化している。それが問題なのだ。学校の表向きの目的は子供を教育することだが、それをちゃんと行わせるための圧力はどこからもかからない。だから多くの学校で行われている教育はひどいもので、子供たちは真面目にそれを受けようとはしない。勉強のできる子供たちでさえもだ。ほとんどの時間、全員が、生徒も教師も、決まった動きを機械的に繰り返しているにすぎない。

In my high school French class we were supposed to read Hugo's *Les Miserables*. I don't think any of us knew French well enough to make our way through this enormous book. Like the rest of the class, I just skimmed the Cliff's Notes. When we were given a test on the book, I noticed that the questions sounded odd. They were full of long words that our teacher wouldn't have used. Where had these questions come from? From the Cliff's Notes, it turned out. The teacher was using them too. We were all just pretending.

高校のフランス語の授業では、一応ユーゴーの「レ・ミゼラブル」を読むことになっていた。でもあの分厚い本を読めるほどフランス語を知っていた生徒は一人もいなかったんじゃないかと思う。他の生徒と同じように、私もテスト対策本を流し読みしただけだった。この本に関するテストを受けたとき、私は問題が妙なことに気付いた。先生が授業で使ったこともない長い単語がいっぱいだったんだ。どこからそんな単語が来たんだろう。やっぱりテスト対策本からだったのさ。先生もそれを使っていたんだ。先生は授業をするふりを、私たちは授業を受けているふりをしていただけだったんだ。

There are certainly great public school teachers. The energy and imagination of my fourth grade teacher, Mr. Mihalko, made that year something his students still talk about, thirty years later. But teachers like him were individuals swimming upstream. They couldn't fix the system.

確かに、公立学校にも立派な教師はいる。私が 4 年生の時のミハルコ先生のエネルギーと想像力は、30 年経った今でも当時の生徒たちの語り草になっている。でもそういう教師は、ひとりぼっちで流れに逆らって泳いでいるだけだ。制度を変えることはできない。

In almost any group of people you'll find hierarchy. When groups of adults form in the real world, it's generally for some common purpose, and the leaders end up being those who are best at it. The problem with most schools is, they have no purpose. But hierarchy there must be. And so the kids make one out of nothing.ほとんどどんな集団にも階級を見ることができる。実世界で大人が集団を作る場合、それは普通、何か共通の目的のためであり、したがってその目的に一番ふさわしい人物がリーダーとなることが多い。多くの学校の問題は、目的が無いことだ。だが階級は生まれざるを得ない。だから子供たちは何もないところから階級をひねり出すんだ。

We have a phrase to describe what happens when rankings have to be created without any meaningful criteria. We say that the situation *degenerates into a popularity contest*. And that's exactly what happens in most American schools. Instead of depending on some real test, one's rank depends mostly on one's ability to increase one's rank. It's like the court of Louis XIV. There is no external opponent, so the kids become one another's opponents.

これといった基準がないところに順位を付けなければならない場合、何が起きるかという
と、墮落したゲームが始まるのだ。「人気取り競争」とでも呼べばいいだろうか、米国のほとんどの学校で起きているのはまさにそれだ。生徒の順位は、本物のテストではなく、自分がどれだけ順位を上げられるかという能力によって決まる。ルイ 14 世の宮廷のようなものだ。外敵がいらないから、子供たちはお互いを敵とするんだ。

When there is some real external test of skill, it isn't painful to be at the bottom of the hierarchy. A rookie on a football team doesn't resent the skill of the veteran; he hopes to be like him one day and is happy to have the chance to learn from him. The veteran may in turn feel a sense of *noblesse oblige*. And most importantly,

their status depends on how well they do against opponents, not on whether they can push the other down.

外部に能力を試すはっきりしたテストがあれば、一番下の階級にいても苦しくはない。フットボールチームのルーキーはベテランプレーヤーの能力に腹を立てたりはしない。むしろいつかそうなりたいと思うだろうし、先輩に学ぶ機会があれば喜んで学ぶはずだ。先輩も上に立つ者の責務、つまりノブレス・オブリージュを感じるだろう。そしてもっと重要なことは、メンバーの地位はどれだけ対戦相手に対して戦えるかによって決まり、お互いの足をどれだけ引っ張れるかによって決まることはないということだ。

Court hierarchies are another thing entirely. This type of society debases anyone who enters it. There is neither admiration at the bottom, nor *noblesse oblige* at the top. It's kill or be killed.

宮廷の階級は全く違ったものだ。この種の社会は中に入る者全てを悪くする。下の者が上を称賛することもないし、上に立つ者が責任を感じることもない。殺るか殺られるかだ。

This is the sort of society that gets created in American secondary schools. And it happens because these schools have no real purpose beyond keeping the kids all in one place for a certain number of hours each day. What I didn't realize at the time, and in fact didn't realize till very recently, is that the twin horrors of school life, the cruelty and the boredom, both have the same cause.

米国の中学校で作られる社会もまさにそれだ。そうなるのは、子供を毎日特定の時間、一箇所に閉じ込めておく以上の本当の目的を学校が持たないからだ。私が当時気付かなかったことは、いや、ごく最近まで気付かなかったことは、学校生活の 2 つの恐怖、つまり、残酷さと退屈さの根はひとつだったということだ。

The mediocrity of American public schools has worse consequences than just making kids unhappy for six years. It breeds a rebelliousness that actively drives kids away from the things they're supposed to be learning.

米国の公立学校の凡庸さは、子供を 6 年間不幸にするだけでなく、もっと悪い結果をもたらす。本来学ぶべき事柄に対して背を向けるような反抗心をむしろ育てるのだ。

Like many nerds, probably, it was years after high school before I could bring myself to read anything we'd been assigned then. And I lost more than books. I mistrusted words like "character" and "integrity" because they had been so

debased by adults. As they were used then, these words all seemed to mean the same thing: obedience. The kids who got praised for these qualities tended to be at best dull-witted prize bulls, and at worst facile schmoozers. If that was what character and integrity were, I wanted no part of them.

たぶん他のオタクと同じように、私も、当時の課題図書を真面目に読む気になったのは、高校を卒業して何年も経ってからだった。失ったものは本だけではない。「人格者」だとか「高潔さ」といった単語は、大人たちによってあまりに貶められていたために、私はそれらを信用していなかった。当時そういう単語は服従と同じ意味で使われているように思えた。こういう単語を使って褒められる子供は、良くてぼーっとして真面目だけが取り柄のような子供、悪ければ狡猾なごますり野郎だった。それが人格者だとか高潔だとかいうなら、私はそんなものにはなりたくないと思っていた。

The word I most misunderstood was "tact." As used by adults, it seemed to mean keeping your mouth shut. I assumed it was derived from the same root as "tacit" and "taciturn," and that it literally meant being quiet. I vowed that I would never be tactful; they were never going to shut me up. In fact, it's derived from the same root as "tactile," and what it means is to have a deft touch. Tactful is the opposite of clumsy. I don't think I learned this until college.

私が一番誤解していたのは「気が利く」(tact)という言葉だった。大人が気を利かせなさいというとき、それは黙っていなさいということのように思えた。だから、その言葉が「無言の」(tacit)や「無口な」(taciturn)と同じ語源から来ていて、文字通り静かにしていることを意味するんだと思っていた。気を利かせるような人間にはなるまいと決心した。黙っているなんてまっぴらだっけね。実はその言葉は「触覚の」(tactile)と同じ語源から来ていて、器用な感覚を持つことを意味していたんだ。気が利くというのは不器用(clumsy)の反対語だ。このことを知ったのは確か大学のころだったと思う。

Nerds aren't the only losers in the popularity rat race. Nerds are unpopular because they're distracted. There are other kids who deliberately opt out because they're so disgusted with the whole process.

オタクだけが人気取り競争の敗者ではない。オタクは他に気を取られることで不人気になる。それとは別に、このシステムにあまりに幻滅して、自分から逃れようとした子供たちがいる。

Teenage kids, even rebels, don't like to be alone, so when kids opt out of the

system, they tend to do it as a group. At the schools I went to, the focus of rebellion was drug use, specifically marijuana. The kids in this tribe wore black concert t-shirts and were called "freaks."

ティーンの子供たちは独りでいることを好まない。不良だってそう。だから子供がシステムから抜けることを決めたら集団で抜けたがる。私が通った学校では、不良たちの関心はドラッグ、特にマリファナだった。このグループの子供たちは黒いコンサート T シャツを着て、「フリーク」と呼ばれていた。

Freaks and nerds were allies, and there was a good deal of overlap between them. Freaks were on the whole smarter than other kids, though never studying (or at least never appearing to) was an important tribal value. I was more in the nerd camp, but I was friends with a lot of freaks.

フリークとオタクは同盟であり、共通項を多く持っていた。フリークは全体的に他の子供より賢かったが、勉強しないこと(少なくとも勉強していないように見せること)が彼らにとって重要な価値だった。私はどちらかというとおタク側にいたが、フリークの友人もたくさんいた。

They used drugs, at least at first, for the social bonds they created. It was something to do together, and because the drugs were illegal, it was a shared badge of rebellion.

彼らは、少なくとも最初は、自分たちの作ったグループの結束のためにドラッグを使った。一緒に何かをやるということでもあったし、非合法なことをやることで、反逆者の勲章を共有するということでもあった。

I'm not claiming that bad schools are the whole reason kids get into trouble with drugs. After a while, drugs have their own momentum. No doubt some of the freaks ultimately used drugs to escape from other problems-- trouble at home, for example. But, in my school at least, the reason most kids *started* using drugs was rebellion. Fourteen-year-olds didn't start smoking pot because they'd heard it would help them forget their problems. They started because they wanted to join a different tribe.

私は子供がドラッグをやるのは学校が悪いからだと言っているわけじゃない。結局のところ、ドラッグはそれ自体の推進力を持っている。最終的に、フリークは間違いなくドラッグを家でのトラブルやいろいろなことから逃れるために使っていたが、少なくとも私の学校では、子供たちがドラッグを使う理由は反抗心からだ。14歳の子供たちはマリファナを使うと嫌なことを忘れられると聞いて始めたわけじゃない。単に違うグループに属した

くて始めただけなんだ。

Misrule breeds rebellion; this is not a new idea. And yet the authorities still for the most part act as if drugs were themselves the cause of the problem.

間違った統治は反抗を育てる。昔から言われていることだ。だがエライ人たちは、ドラッグはそれ自体が問題の原因であるといった見方をいまだに取り続けている。

The real problem is the emptiness of school life. We won't see solutions till adults realize that. The adults who may realize it first are the ones who were themselves nerds in school. Do you want your kids to be as unhappy in eighth grade as you were? I wouldn't. Well, then, is there anything we can do to fix things? Almost certainly. There is nothing inevitable about the current system. It has come about mostly by default.

真の問題は、学校生活の空虚さだ。大人がこれを認識しない限り解決はあり得ない。これに最初に気付くのは、自分も学校でオタクだった大人たちだろう。自分の子供に、自分がそうであったのと同じように不幸な第 8 学年を送らせたいかい? 私は嫌だ。それなら、これを正すために私たちができることはあるだろうか。あるはずだ。現在の制度に必然的なものは何もない。何もしないからそうになっているだけなんだ。

Adults, though, are busy. Showing up for school plays is one thing. Taking on the educational bureaucracy is another. Perhaps a few will have the energy to try to change things. I suspect the hardest part is realizing that you can.

でも、大人は忙しい。官僚的教育制度に切り込むのは、学校の文化祭を見に行くのとはわけが違う。制度を変えようとするだけの勢力を持っている人はほとんどいないだろう。一番難しいのは、自分にそれができると気付くことじゃないだろうか。

Nerds still in school should not hold their breath. Maybe one day a heavily armed force of adults will show up in helicopters to rescue you, but they probably won't be coming this month. Any immediate improvement in nerds' lives is probably going to have to come from the nerds themselves.

まだ学校にいるオタクは、息を潜めているべきじゃない。いつか重武装した大人たちがヘリコプターで君たちを助けに来てくれるかもしれないが、それは今月ではないだろう。今、状況を改善したいのなら、君たちが自ら行動を起こすしかない。

Merely understanding the situation they're in should make it less painful. Nerds aren't losers. They're just playing a different game, and a game much closer to the one played in the real world. Adults know this. It's hard to find successful adults now who don't claim to have been nerds in high school.

自分たちのいる状況を理解するだけでも、苦痛を和らげることになるはずだ。オタクは負け組じゃない。単に違うゲームをやっているだけなんだ。それも、実社会により近いゲームを。大人もこれを知っている。現在成功している大人で、中学高校でオタクじゃなかったと言い切れる人は滅多にいない。

It's important for nerds to realize, too, that school is not life. School is a strange, artificial thing, half sterile and half feral. It's all-encompassing, like life, but it isn't the real thing. It's only temporary, and if you look, you can see beyond it even while you're still in it.

また、オタクは、学校が人生の全てではないと理解することも重要だ。学校は奇妙で人工的な、空虚さと凶暴さの混じり合った場所だ。それに取り囲まれていると、まるでそれが人生のように思えてしまう。でも、それは本物の人生じゃない。ほんの一時的なものだし、よく見れば、中にいる時からその向こうを見通せるはずだ。

If life seems awful to kids, it's neither because hormones are turning you all into monsters (as your parents believe), nor because life actually is awful (as you believe). It's because the adults, who no longer have any economic use for you, have abandoned you to spend years cooped up together with nothing real to do. *Any* society of that type is awful to live in. You don't have to look any further to explain why teenage kids are unhappy.

子供にとって人生がひどいものだとしたら、それは(親が信じているように)ホルモンが子供を怪物にしているわけでもなく、(あなたが信じているように)人生が実際にひどいものであるわけでもない。子供に稼がせることができなくなった大人たちが、何もすることがない場所に何年もあなたたちを閉じ込めて放っておいたからだ。そんな社会はどこであれ悲惨なものになる。ティーンエイジの子供が不幸な理由としてこれ以上の説明はいらないだろう。

I've said some harsh things in this essay, but really the thesis is an optimistic one-- that several problems we take for granted are in fact not insoluble after all. Teenage kids are not inherently unhappy monsters. That should be encouraging news to kids and adults both.

このエッセイの中でいくつか厳しいことも述べたが、私の見方は楽観的だ。どうにもなら

ないと思える問題でも、実は解決できるんだ。ティーンエイジの子供たちは、生まれつき不遇な怪物であったわけではない。これは、子供にとっても大人にとっても、勇気づけられる事実ではないか。

Thanks to Sarah Harlin, Trevor Blackwell, Robert Morris, Eric Raymond, and Jackie Weicker for reading drafts of this essay, and Maria Daniels for scanning photos.

Chapter 2 HACKERS AND PAINTERS

第2章 ハッカーと画家

May 2003

(This essay is derived from a guest lecture at Harvard, which incorporated an earlier talk at Northeastern.)

(このエッセイはハーバード大学での特別講義と、それに先立つノースイースタン大学での講演を基にしている。)

When I finished grad school in computer science I went to art school to study painting. A lot of people seemed surprised that someone interested in computers would also be interested in painting. They seemed to think that hacking and painting were very different kinds of work-- that hacking was cold, precise, and methodical, and that painting was the frenzied expression of some primal urge.

大学院で計算機科学を専攻した後、私は絵画を学ぶために美術学校に入った。コンピュータに興味を持つような人間が絵を描くことにも興味を持っていると聞いて、驚く人も多い。どうやらハッキングと絵を描くことは全然違うものだと思われるらしい。ハッキングは冷たく精密で几帳面なものであるのに対し、絵を描くことは何か原始的な衝動に駆られた表現行為だと考えられているようだ。

Both of these images are wrong. Hacking and painting have a lot in common. In fact, of all the different types of people I've known, hackers and painters are among the most alike.

そのイメージはどちらも正しくない。ハッキングと絵を描くことにはたくさんの共通点がある。実際、私が知っているあらゆる種類の人々のうちで、ハッカーと画家が一番よく似ている。

What hackers and painters have in common is that they're both makers. Along with composers, architects, and writers, what hackers and painters are trying to do is make good things. They're not doing research per se, though if in the course of trying to make good things they discover some new technique, so much the better.

ハッカーと画家に共通することは、どちらもものを創る人間だということだ。作曲家や建築家や作家と同じように、ハッカーと画家がやろうとしているのは、良いものを創ることだ。良いものを創ろうとする過程で新しいテクニックを発見することがあり、それ

はそれで良いことだが、いわゆる研究活動とはちょっと違う。

I've never liked the term "computer science." The main reason I don't like it is that there's no such thing. Computer science is a grab bag of tenuously related areas thrown together by an accident of history, like Yugoslavia. At one end you have people who are really mathematicians, but call what they're doing computer science so they can get DARPA grants. In the middle you have people working on something like the natural history of computers-- studying the behavior of algorithms for routing data through networks, for example. And then at the other extreme you have the hackers, who are trying to write interesting software, and for whom computers are just a medium of expression, as concrete is for architects or paint for painters. It's as if mathematicians, physicists, and architects all had to be in the same department.

私は「計算機科学」という用語がどうにも好きになれない。一番大きな理由は、そもそもそんなものは存在しないからだ。計算機科学とは、ほとんど関連のない分野が歴史的な偶然からいっしょくたに袋に放り込まれたもので、言ってみればユーゴスラビアみたいなものだ。一方の端では、本当は数学者である人々が、**DARPA** の研究費を得るために計算機科学を名乗っている。真ん中あたりでは、コンピュータの博物学みたいなことをやっている人々がいる。ネットワークのルーティングアルゴリズムの振る舞いを調べたりとか、そういうことだ。そして、反対側の端っこには、ハッカーたちがいる。面白いソフトを書こうとしている者たちだ。彼らにとってはコンピュータは単なる表現の媒体にすぎない。建築家にとってのコンクリートが、また画家にとっての絵の具がそうであるように。これじゃまるで、数学者と物理学者と建築家をひとつの学科に押し込めているみたいだ。

Sometimes what the hackers do is called "software engineering," but this term is just as misleading. Good software designers are no more engineers than architects are. The border between architecture and engineering is not sharply defined, but it's there. It falls between what and how: architects decide what to do, and engineers figure out how to do it.

ハッカーがやっていることは「ソフトウェア工学」と呼ばれることもあるが、この用語も誤解を助長するだけだ。優れたソフトウェア設計者は、建築家がエンジニアではないのと同じように、エンジニアではない。もちろん建築とエンジニアリングの境界ははっきりと定められているわけじゃないけれど、確かに存在する。それは「何を」と「どうやって」の間にある。建築家は何をするかを決め、エンジニアはそれをどうやってするかを考え出

すのだ。

What and how should not be kept too separate. You're asking for trouble if you try to decide what to do without understanding how to do it. But hacking can certainly be more than just deciding how to implement some spec. At its best, it's creating the spec-- though it turns out the best way to do that is to implement it.

「何を」と「どうやって」をあまり分け過ぎるのは良くない。どうやればできるかを理解せずに何をするかを決めるのは間違いのもとだ。でも、ハッキングには確かに、ある仕様をどうやって実装するか決めること以上のものがある。ハッキングの最良の形態とは、仕様を創ることだ。ただ、仕様を創る一番の方法はそれを実装することだ、ということにすぎない。

Perhaps one day "computer science" will, like Yugoslavia, get broken up into its component parts. That might be a good thing. Especially if it meant independence for my native land, hacking.

たぶん「計算機科学」は、ユーゴスラビアがそうなったように、いつかそれを構成している部分ごとにばらばらになるだろう。それは良いことなんだろうと思う。特に、それが私の故郷でもあるハッカー国の独立を意味するのなら。

Bundling all these different types of work together in one department may be convenient administratively, but it's confusing intellectually. That's the other reason I don't like the name "computer science." Arguably the people in the middle are doing something like an experimental science. But the people at either end, the hackers and the mathematicians, are not actually doing science.

これらの全然違う業種をひとつの学科にまとめておくのは、管理する側にとっては便利なのかもしれないが、知的な混乱をもたらす。「計算機科学」という言葉を私が嫌うもうひとつの理由がそれだ。真ん中にいる人々は、まあ実験科学をやっていると言えなくはないかもしれない。しかし両端にいる人々、ハッカーと数学者は、科学をやっているわけじゃないんだ。

The mathematicians don't seem bothered by this. They happily set to work proving theorems like the other mathematicians over in the math department, and probably soon stop noticing that the building they work in says "computer science" on the outside. But for the hackers this label is a problem. If what they're

doing is called science, it makes them feel they ought to be acting scientific. So instead of doing what they really want to do, which is to design beautiful software, hackers in universities and research labs feel they ought to be writing research papers.

数学者はきっとそんなことは気にしないだろう。彼らは数学科の数学者たちと同じように定理を証明することに夢中になって、自分のいる建物に「計算機科学」の看板が付いていることなどたぶんすぐに忘れてしまう。でもハッカーにとってはこの看板は問題だ。「科学」と呼ばれると、ハッカーたちは科学的に振る舞わなくちゃならないような気になってしまふ。そして大学や研究所にいるハッカーたちは、彼らが本当にやりたいこと、つまり美しいソフトウェアをデザインすることではなしに、研究論文を書かなくちゃいけないような気になってしまうんだ。

In the best case, the papers are just a formality. Hackers write cool software, and then write a paper about it, and the paper becomes a proxy for the achievement represented by the software. But often this mismatch causes problems. It's easy to drift away from building beautiful things toward building ugly things that make more suitable subjects for research papers.

運がよければ、論文は形を整えるためだけのもので済むだろう。ハッカーはクールなソフトを書いて、そしてそれについての論文を書く。ソフトウェアそのものを成果とする代わりに論文を成果とすることにするんだ。でもこのミスマッチは往々にして問題となる。美しいものを創るのではなしに、醜いけれど論文の題目にはなりやすいものを造るほうに引き込まれやすいからだ。

Unfortunately, beautiful things don't always make the best subjects for papers. Number one, research must be original-- and as anyone who has written a PhD dissertation knows, the way to be sure that you're exploring virgin territory is to stake out a piece of ground that no one wants. Number two, research must be substantial-- and awkward systems yield meatier papers, because you can write about the obstacles you have to overcome in order to get things done. Nothing yields meaty problems like starting with the wrong assumptions. Most of AI is an example of this rule; if you assume that knowledge can be represented as a list of predicate logic expressions whose arguments represent abstract concepts, you'll have a lot of papers to write about how to make this work. As Ricky Ricardo used to say, "Lucy, you got a lot of explaining to do."

残念なことに、美しいものは論文になりやすいとは限らない。第一に、研究は独創的でなければならない。そして、博士論文を書いた経験のある人なら誰もが知っているように、

確実に処女地を探し当てて開拓する一番良い方法は、誰も向かおうとはしない場所へ向かうことだ。第二に、研究にはある程度の量的なまとまりが必要だ。そして、へんてこりんなシステムであるほど、たくさんの論文が書ける。そいつを動かすために乗り越えなければならなかったいろんな障害について書けるからね。論文の数を増やす最良の方法は、間違った仮定から出発することだ。人工知能の研究の多くがこれに当てはまる。知識は抽象概念を引数に取る述語論理式の羅列で表現できる、と仮定して始めれば、それを動かすためにたくさんの論文を書くことになるだろう。リッキー・リカルドが言ったように、「ルーシー、君はたくさん説明することがあるね」ってなわけだ。

The way to create something beautiful is often to make subtle tweaks to something that already exists, or to combine existing ideas in a slightly new way. This kind of work is hard to convey in a research paper.

何か美しいものを創るということは、しばしば既にあるものに微妙な改良を加えたり、既にある考えを少しだけ新しい方法で組み合わせたりすることによってなされる。この種の仕事を研究論文にするのはとても難しい。

So why do universities and research labs continue to judge hackers by publications? For the same reason that "scholastic aptitude" gets measured by simple-minded standardized tests, or the productivity of programmers gets measured in lines of code. These tests are easy to apply, and there is nothing so tempting as an easy test that kind of works.

じゃあ、大学や研究所はどうしてハッカーを論文数で判断しようとするんだろうか。「学習への適性」が考えの狭い共通テストで測られたり、プログラマの生産性がコードの行数で測られたりするのと同じ理由だ。これらのテストは簡単に適用できる。そして、簡単に適用できてとりあえず使えるテストほど便利なものはない。

Measuring what hackers are actually trying to do, designing beautiful software, would be much more difficult. You need a good [sense of design](#) to judge good design. And there is no correlation, except possibly a [negative](#) one, between people's ability to recognize good design and their confidence that they can.

ハッカーが実際にやろうとしていること、すなわち美しいソフトウェアをデザインすることは、ずっと測るのが難しい。良いデザインを認識するためには、デザインに対するセンスが必要だ。ある人が良いデザインを認識する能力と、その人が良いデザインを認識できると考えている自信の間には、何の相関もないか、あったとしても、負の相関しか

ないだろう。

The only external test is time. Over time, beautiful things tend to thrive, and ugly things tend to get discarded. Unfortunately, the amounts of time involved can be longer than human lifetimes. Samuel Johnson said it took a hundred years for a writer's reputation to converge. You have to wait for the writer's influential friends to die, and then for all their followers to die.

唯一の外部のテストは時間だ。時間が経てば、美しいものは生き残り、醜いものは次第に淘汰されてゆくだろう。それにかかる時間は、残念ながら、人間の一生よりも長い。サミュエル・ジョンソンは、作家の評価が収束するまでには 100 年かかると言った。作家の影響下にあった友人たちが死に、さらにその信奉者たちが死に絶えるのを待たねばならないからだ。

I think hackers just have to resign themselves to having a large random component in their reputations. In this they are no different from other makers. In fact, they're lucky by comparison. The influence of fashion is not nearly so great in hacking as it is in painting.

ハッカーは自分の評価の中にランダムな要素がたくさん紛れ込むことを甘受しなければならないだろう。その点では、他のもの創りの人々と変わらない。むしろ、他のもの創りよりも幸運かもしれない。ハッキングに対する流行の影響は、絵画に対するそれほどには大きくないからだ。

There are worse things than having people misunderstand your work. A worse danger is that you will yourself misunderstand your work. Related fields are where you go looking for ideas. If you find yourself in the computer science department, there is a natural temptation to believe, for example, that hacking is the applied version of what theoretical computer science is the theory of. All the time I was in graduate school I had an uncomfortable feeling in the back of my mind that I ought to know more theory, and that it was very remiss of me to have forgotten all that stuff within three weeks of the final exam.

自分の仕事を他人に誤解されるよりも悪いことがある。それは、あなた自身が自分の仕事を誤解することだ。アイディアを探す時、人は関連した分野を見にゆく。あなたが計算機科学科にいたとしたら、ハッキングは理論計算機科学に対する応用だとか思ってしまったりかもしれない。私は大学院にいたころずっと、もっと理論を知らなくちゃならないという思

いが心の底にあって、居心地の悪い思いをしていた。期末試験から 3 週間後にはすっかり全部忘れてしまうことを、ずいぶん後ろめたく思ったものだ。

Now I realize I was mistaken. Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry. You need to know how to calculate time and space complexity and about Turing completeness. You might also want to remember at least the concept of a state machine, in case you have to write a parser or a regular expression library. Painters in fact have to remember a good deal more about paint chemistry than that.

でも私は間違っていたんだ。ハッカーは、画家が絵の具に関する化学を理解するのと同程度に計算理論を理解していればいい。時間的および空間的な複雑度の計算法と、パーザを書くときのために、状態機械の概念くらいを知っておけばいいだろう。画家は絵の具の化学について、そんなことよりもっとずっとたくさんのことを覚えなければならないんだ。

I've found that the best sources of ideas are not the other fields that have the word "computer" in their names, but the other fields inhabited by makers. Painting has been a much richer source of ideas than the theory of computation. 私は、アイディアの最高の源は「コンピュータ」が名前に付いている分野じゃなくて、ものを創る人々が住んでいる他の分野にあるということに気付いた。絵画は計算理論よりもずっと豊富なアイディアを提供してくれたんだ。

For example, I was taught in college that one ought to figure out a program completely on paper before even going near a computer. I found that I did not program this way. I found that I liked to program sitting in front of a computer, not a piece of paper. Worse still, instead of patiently writing out a complete program and assuring myself it was correct, I tended to just spew out code that was hopelessly broken, and gradually beat it into shape. Debugging, I was taught, was a kind of final pass where you caught typos and oversights. The way I worked, it seemed like programming consisted of debugging.

例えば大学で私は、コンピュータに手を触れる前に紙の上でプログラムを完全に理解しなければならないと教わった。でも私はそういうふうにはプログラムできなかった。私が好んだやり方は、紙の前ではなく、コンピュータの前に座ってプログラミングすることだった。しかも、辛抱強く全てのプログラムを書き上げて正しいことを確認するなんてことはせずに、めちゃくちゃなコードをおっぴろげて、それを次第に形にしてゆくのだった。デバッグとは書き間違いや見逃しをつかまえる最終段階の工程だと教わったけれど、実際に

私がやっていたのは、プログラミングそのものがデバッグという具合だった。

For a long time I felt bad about this, just as I once felt bad that I didn't hold my pencil the way they taught me to in elementary school. If I had only looked over at the other makers, the painters or the architects, I would have realized that there was a name for what I was doing: sketching. As far as I can tell, the way they taught me to program in college was all wrong. You should figure out programs as you're writing them, just as writers and painters and architects do. そのことで私はずいぶんと長い間、引け目を感じていた。ちょうど小学校で教わった鉛筆の持ち方と違う持ち方をしていることに引け目を感じていたのと同じように。他のものを創る人々、画家や建築家がどうやっているかを見れば、私は自分のやっていることにちゃんと名前がついていると気付いていただろう。スケッチだ。私の知る限り、私が大学で教わったプログラミングのやり方は全部間違っていた。作家や画家や建築家が創りながら作品を理解してゆくのと同じで、プログラマはプログラムを書きながら理解してゆくべきなんだ。

Realizing this has real implications for software design. It means that a programming language should, above all, be malleable. A programming language is for [thinking](#) of programs, not for expressing programs you've already thought of. It should be a pencil, not a pen. Static typing would be a fine idea if people actually did write programs the way they taught me to in college. But that's not how any of the hackers I know write programs. We need a language that lets us scribble and smudge and smear, not a language where you have to sit with a teacup of types balanced on your knee and make polite conversation with a strict old aunt of a compiler.

これを知ることは、ソフトウェアの設計に大きな意味を持つ。まず、プログラミング言語は何よりも柔軟でなければならないということが言える。プログラミング言語はプログラムを考えるためのものであって、既に考えたプログラムを書き下すためのものじゃない。それはペンではなく鉛筆であるべきなんだ。静的型付けは、私が大学で教わったようにプログラムするなら良い考えだと思う。でも私の知るハッカーたちはそんなふうにはプログラムしない。私たちに必要なのは、落書きしたりぼかしたり塗りつぶしたりできる言語であって、型の紅茶茶碗を膝に置いて、落とさないようバランスをとりながら、作法にうるさいコンパイラお婆さんとお上品な会話をするような言語じゃない。

While we're on the subject of static typing, identifying with the makers will save us from another problem that afflicts the sciences: math envy. Everyone in the sciences secretly believes that mathematicians are smarter than they are. I think mathematicians also believe this. At any rate, the result is that scientists tend to make their work look as mathematical as possible. In a field like physics this probably doesn't do much harm, but the further you get from the natural sciences, the more of a problem it becomes.

静的型付けの話が出たついでにもうひとつ、科学が抱えている問題で、もの創りの人々ならば避け得るものを挙げておこう。数学に対する羨望だ。科学にかかわる人は誰しも、数学者は自分より聡明だという思いを密かに抱いている。そして、数学者たちもそう信じているんじゃないだろうか。結果として科学者たちは、程度の差こそあれ、自分の仕事をなるべく数学っぽく見せようとしがちだ。物理学のような分野ではこのことはたいてい悪影響を及ぼさないだろう。しかし自然科学から離れれば離れるほど、これはより大きな問題となってくる。

A page of formulas just looks so impressive. (Tip: for extra impressiveness, use Greek variables.) And so there is a great temptation to work on problems you can treat formally, rather than problems that are, say, important.

数式で埋め尽くされたページは、それはそれは格好良く見える。(もっと格好良くしたいのなら変数にギリシャ文字を使うといい)。だから、重要な問題よりも形式的に扱える問題のほうに取り組みたいという誘惑はとても強い。

If hackers identified with other makers, like writers and painters, they wouldn't feel tempted to do this. Writers and painters don't suffer from math envy. They feel as if they're doing something completely unrelated. So are hackers, I think. ハッカーを作家や画家といった他のもの創りと同列に並べて考えるなら、そんな誘惑は感じないだろう。作家や画家は数学を羨んだりしない。数学とは全然関係ないことをやっていると思っているだろう。ハッカーもそう感じるべきだと、私は思う。

If universities and research labs keep hackers from doing the kind of work they want to do, perhaps the place for them is in companies. Unfortunately, most companies won't let hackers do what they want either. Universities and research labs force hackers to be scientists, and companies force them to be engineers. 大学や研究所がハッカーに本当にやりたいことをやらせてくれないとしたら、企業に行く

しかないのだろうか。残念ながら、多くの企業はやっぱりハッカーにやりたいことをやらせてはくれない。大学や研究所はハッカーに科学者たれとを強要するが、企業はハッカーにエンジニアたれとを強要するからだ。

I only discovered this myself quite recently. When Yahoo bought Viaweb, they asked me what I wanted to do. I had never liked the business side very much, and said that I just wanted to hack. When I got to Yahoo, I found that what hacking meant to them was implementing software, not designing it. Programmers were seen as technicians who translated the visions (if that is the word) of product managers into code.

私はこのことにはごく最近になるまで気付かなかった。**Viaweb** が **Yahoo!** に買収された時、彼らは私に何をやりたいか尋ねた。私はビジネスには全然興味がなかったので、ハックしたいんだと答えた。**Yahoo!** に入ってみたら、そこではハックするということはソフトウェアを実装するということで、デザインするということではないということが分かった。プログラマは、プロダクトマネージャのビジョンとかいったものをコードへと翻訳する技師と見なされていたのだ。

This seems to be the default plan in big companies. They do it because it decreases the standard deviation of the outcome. Only a small percentage of hackers can actually design software, and it's hard for the people running a company to pick these out. So instead of entrusting the future of the software to one brilliant hacker, most companies set things up so that it is designed by committee, and the hackers merely implement the design.

どうも大企業ではそれが普通らしい。そういうふうになれば、出力のばらつきを抑えることができるからだ。ハッカーのうち実際にソフトウェアをデザインできる能力のある人間はほんの一握りであって、企業を経営している人々がそういうハッカーを見つけ出すのはとても難しい。だから多くの企業では、ソフトウェアの未来を一人の才能あふれるハッカーに託すのではなく、委員会によって設計し、ハッカーはそれをただ実装するだけという仕組みを作るんだ。

If you want to make money at some point, remember this, because this is one of the reasons startups win. Big companies want to decrease the standard deviation of design outcomes because they want to avoid disasters. But when you damp oscillations, you lose the high points as well as the low. This is not a problem for big companies, because they don't win by making great products. Big companies win by sucking less than other big companies.

あなたがいつか財をなしたいと思っているなら、このことを覚えておくといい。なぜならこれは、ベンチャー企業が勝つ理由の一つだからだ。大企業が出力のばらつきを抑えたがるのは、大失敗を避けたいからだ。でもばらつきをなくすと、低い点は消えるけれど高い点も消えてしまう。大企業ではそれは問題じゃない。大企業はすごい製品を作ることで勝つのではなく、他の企業よりも下手を打たないことで勝つからだ。

So if you can figure out a way to get in a design war with a company big enough that its software is designed by product managers, they'll never be able to keep up with you. These opportunities are not easy to find, though. It's hard to engage a big company in a design war, just as it's hard to engage an opponent inside a castle in hand to hand combat. It would be pretty easy to write a better word processor than Microsoft Word, for example, but Microsoft, within the castle of their operating system monopoly, probably wouldn't even notice if you did.

だから、ソフトウェアがプロダクトマネージャたちによって設計されるような大企業とデザインで勝負する方法を見つければ、彼らは絶対あなたには勝てない。でもそういう機会を見つけるのは簡単ではない。大企業をデザインでの勝負の土俵に引っ張り出すのは、城の中にいる相手に一对一の勝負に応じさせるのと同じくらい難しい。例えば、マイクロソフトの Word より良いワードプロセッサを書くのはたいして難しくないだろう。でも独占 OS の城の中にいる彼らは、おそらくあなたがそれを書いたことに気づきさえしないだろう。

The place to fight design wars is in new markets, where no one has yet managed to establish any fortifications. That's where you can win big by taking the bold approach to design, and having the same people both design and implement the product. Microsoft themselves did this at the start. So did Apple. And Hewlett-Packard. I suspect almost every successful startup has.

デザインで勝負に打って出る場所は、誰も要塞を築いていない新しいマーケットだ。そこでなら、デザインに大胆なアプローチを採り入れ、そして同一人物がデザインと実装を受け持つことで、大きく勝つことができる。マイクロソフトだって最初はそこから始まったんだ。アップルもそうだし、ヒューレットパッカードもそうだ。おそらくどんな成功したベンチャーもそうだと思う。

So one way to build great software is to start your own startup. There are two problems with this, though. One is that in a startup you have to do so much besides write software. At Viaweb I considered myself lucky if I got to hack a

quarter of the time. And the things I had to do the other three quarters of the time ranged from tedious to terrifying. I have a benchmark for this, because I once had to leave a board meeting to have some cavities filled. I remember sitting back in the dentist's chair, waiting for the drill, and feeling like I was on vacation.

すごいソフトを書く方法のひとつは、自分でベンチャー企業を作ることだ。でもそれには2つ問題がある。ひとつは、ベンチャー企業を作ると、ソフトを書く以外のことをたくさんやらなくちゃならないということだ。Viaweb では、私は 1/4 の時間をハッキングに充てることができたらしいほうだった。そして 3/4 の時間にしなくちゃならなかったのは、うんざりするかぞっとするようなことばかりだった。これにはベンチマークがある。一度私は虫歯を治すために取締役会を抜けなくちゃならなくて、歯医者椅子に座ってドリルが近づいてくるのを待ちながら、なんて素晴らしい休憩時間だろうと感じたものだ。

The other problem with startups is that there is not much overlap between the kind of software that makes money and the kind that's interesting to write. Programming languages are interesting to write, and Microsoft's first product was one, in fact, but no one will pay for programming languages now. If you want to make money, you tend to be forced to work on problems that are too nasty for anyone to solve for free.

ベンチャーのもうひとつの問題は、書くのが面白いソフトが金になるソフトであるということが滅多にないということだ。プログラミング言語を書くのは面白いし、実際、マイクロソフトの最初の製品はそれだったわけだが、今となっては誰もプログラミング言語には金を出さない。金を儲けようと思ったら、誰もただではやりたがらないような危険な問題に取り組まざるを得なくなる。

All makers face this problem. Prices are determined by supply and demand, and there is just not as much demand for things that are fun to work on as there is for things that solve the mundane problems of individual customers. Acting in off-Broadway plays just doesn't pay as well as wearing a gorilla suit in someone's booth at a trade show. Writing novels doesn't pay as well as writing ad copy for garbage disposals. And hacking programming languages doesn't pay as well as figuring out how to connect some company's legacy database to their Web server. 実は、これはものを創る人なら誰しも直面する問題だ。価格は需要と供給で決まる。やっついて楽しい仕事には、顧客のありふれた問題を解決するような仕事ほどには需要がない。オフブロードウェイの芝居に出ても、トレードショーのブースでゴリラの着ぐるみを着るほどには稼げない。小説を書いても、ディスポーザの宣伝文句を書くほどは稼げない。そして、プログラミング言語をハックしていても、顧客企業の古いデータベースとそこの Web

サーバをどうやってつなげるかを考えるほどの金にはならないというわけだ。

I think the answer to this problem, in the case of software, is a concept known to nearly all makers: the day job. This phrase began with musicians, who perform at night. More generally, it means that you have one kind of work you do for money, and another for love.

ソフトウェアに関してのこの問題への解答は、実は他のもの創りの人々には既に知られている。昼間の仕事(day job)というやつだ。この言葉はミュージシャンの間で発生した。彼らは夜に演奏するからだ。より一般的に言えば、生活費のために一つの仕事を、愛のためにもう一つの仕事をするということだ。

Nearly all makers have day jobs early in their careers. Painters and writers notoriously do. If you're lucky you can get a day job that's closely related to your real work. Musicians often seem to work in record stores. A hacker working on some programming language or operating system might likewise be able to get a day job using it. [\[1\]](#)

ほとんど全てのものを創る人たちは、駆け出しの頃には昼間の仕事を持っていた。画家と小説家は特にそうだ。運が良ければ、本業と関係が深い昼間の仕事を見つけることができるだろう。ミュージシャンはよくレコード店で働いている。プログラミング言語や OS をハックしているハッカーは、それらを使う昼間の仕事を見つけることができるかもしれない。

[\[1\]](#)

When I say that the answer is for hackers to have day jobs, and work on beautiful software on the side, I'm not proposing this as a new idea. This is what open-source hacking is all about. What I'm saying is that open-source is probably the right model, because it has been independently confirmed by all the other makers.

昼間の仕事を持ち、美しいソフトウェアを別の時間を書くということがハッカーへの答えだと言ったが、これは目新しいことでもなんでもない。オープンソースのハッキングとはそういうものなんだ。オープンソースのモデルに間違いはあるまい。なぜならそれは元々、他のもの創りの人々によって実証されたものなんだからね。

It seems surprising to me that any employer would be reluctant to let hackers work on open-source projects. At Viaweb, we would have been reluctant to hire

anyone who didn't. When we interviewed programmers, the main thing we cared about was what kind of software they wrote in their spare time. You can't do anything really well unless you love it, and if you love to hack you'll inevitably be working on projects of your own. [2]

雇っているハッカーがオープンソースプロジェクトに参加するのを嫌がる企業があるというのは、私にとっては驚きだ。Viaweb では、私たちはむしろそういうプロジェクトに参加していない人を雇うのを避けたものだ。プログラマを面接する時、私たちが一番に知りたかったのは、応募者が余暇にどんなプログラムを書いているかということだった。何かに愛がなければ、それを本当にうまくやることはできないし、ハッキングに愛があれば、いずれ自分のプロジェクトを立ち上げずにはおれないからだ。[2]

Because hackers are makers rather than scientists, the right place to look for metaphors is not in the sciences, but among other kinds of makers. What else can painting teach us about hacking?

ハッカーは科学者よりももの創りに似ているのだから、メタファーを探すのは科学者よりも他のもの創りの分野からのほうがよい。絵を描くことは、ハッキングに対して他にどんなことを教えてくれるだろう。

One thing we can learn, or at least confirm, from the example of painting is how to learn to hack. You learn to paint mostly by doing it. Ditto for hacking. Most hackers don't learn to hack by taking college courses in programming. They learn to hack by writing programs of their own at age thirteen. Even in college classes, you learn to hack mostly by hacking. [3]

絵画の例からひとつ学べることは、少なくとも確認できることは、ハッキングをどうやって学んだらよいかということだ。絵を描くことは、絵を描きながら学ぶ。ハッキングも同じだ。たいていのハッカーは、大学のプログラミングコースを履修してハッキングを学ぶのではない。13歳のころから自分でプログラムを書くことによって学ぶのだ。大学でハッキングを学ぶ人も、ハックしながらハッキングを学ぶことに変わりはない。[3]

Because painters leave a trail of work behind them, you can watch them learn by doing. If you look at the work of a painter in chronological order, you'll find that each painting builds on things that have been learned in previous ones. When there's something in a painting that works very well, you can usually find version 1 of it in a smaller form in some earlier painting.

画家は作品を足跡として残してゆくから、彼らが絵を描きながら学んでゆく様子をうかがい知ることができる。画家の作品を年代順に並べてみれば、それぞれの絵はその前の絵で学んだことの上に創られていることがわかるだろう。絵画上でうまくいったものがある時、通常はそれより前の作品群の中に、より小さな形でのバージョン 1 を見て取ることができる。

I think most makers work this way. Writers and architects seem to as well. Maybe it would be good for hackers to act more like painters, and regularly start over from scratch, instead of continuing to work for years on one project, and trying to incorporate all their later ideas as revisions.

多くのもの創りは同じだと思う。作家と建築家はそうだ。ハッカーもたぶん、画家と同じように振る舞うのがいいんじゃないかと思う。ひとつのプロジェクトを何年も続けるのではなしに、時々ゼロから始め、新しいアイディアを新バージョンとして取り込んでみるんだ。

The fact that hackers learn to hack by doing it is another sign of how different hacking is from the sciences. Scientists don't learn science by doing it, but by doing labs and problem sets. Scientists start out doing work that's perfect, in the sense that they're just trying to reproduce work someone else has already done for them. Eventually, they get to the point where they can do original work. Whereas hackers, from the start, are doing original work; it's just very bad. So hackers start original, and get good, and scientists start good, and get original.

ハッカーがハックしながら学ぶという事実は、ハッキングと科学がどれだけ違うかということを示すもうひとつの手がかりだ。科学者は科学をしながら学ぶのではない。実験と課題をこなしながら学ぶのだ。科学者はまず完璧な仕事から、つまり誰か他の人が既にやったことを再現することから始める。そうしているうちに独自の仕事ができるレベルに達するのだ。一方、ハッカーは最初から独自の仕事をする。ただ最初は下手くそだろう。ハッカーはオリジナルから始めて上手になってゆく。科学者は上手になることから始めてオリジナルになってゆく。

The other way makers learn is from examples. For a painter, a museum is a reference library of techniques. For hundreds of years it has been part of the traditional education of painters to copy the works of the great masters, because copying forces you to look closely at the way a painting is made.

もの創りが学ぶもうひとつの方法は先例から学ぶことだ。画家にとって美術館は技法の先例の宝庫だ。偉大な画家の作品を模写することは、何百年もの間、画家の教育過程の一環となってきた。模写することで、絵がどのように描かれているかを詳しく見るようになるからだ。

Writers do this too. Benjamin Franklin learned to write by summarizing the points in the essays of Addison and Steele and then trying to reproduce them. Raymond Chandler did the same thing with detective stories.

作家も同じことをする。ベンジャミン・フランクリンはアディスンとスティーラーのエッセイを要約し、それを再現しようとする中で書くことを学んだ。レイモンド・チャンドラーは同じことを探偵小説でやった。

Hackers, likewise, can learn to program by looking at good programs-- not just at what they do, but the source code too. One of the less publicized benefits of the open-source movement is that it has made it easier to learn to program. When I learned to program, we had to rely mostly on examples in books. The one big chunk of code available then was Unix, but even this was not open source. Most of the people who read the source read it in illicit photocopies of John Lions' book, which though written in 1977 was not allowed to be published until 1996.

ハッカーも同じように、良いプログラムを見ることでプログラミングを学ぶことができる。プログラムの動作を見るだけでなく、ソースコードを見ることでもだ。あまり知られてはいないが、オープンソースには、プログラミングを学ぶことを容易にするという利点がある。私がプログラミングを学んだころは本に掲載されたサンプルに頼るしかなかった。当時、ソースを見ることができたひとつの大きなソフトウェアは **Unix** だったが、それでさえオープンソースではなかった。**Unix** のソースコードを読んだ人の多くはジョン・ライアンス本の不正なコピーで読んだのではないか。その本は **1977** 年に書かれたが、**1996** 年まで出版を許可されなかった。

Another example we can take from painting is the way that paintings are created by gradual refinement. Paintings usually begin with a sketch. Gradually the details get filled in. But it is not merely a process of filling in. Sometimes the original plans turn out to be mistaken. Countless paintings, when you look at them in xrays, turn out to have limbs that have been moved or facial features that have been readjusted.

絵画から学べるもうひとつの例は、次第に詳細化しながら創ってゆく方法だ。絵画はたいてい、スケッチから始まる。そして次第に細かい部分が埋められてゆく。だがそれは、単に隙間を埋めてゆくだけの過程ではない。ときには元の計画が間違いだったことが分かることもある。X線で見ると、手や足の位置が変更されていたり、表情が変えられていたりしている絵画は数え切れないくらいある。

Here's a case where we can learn from painting. I think hacking should work this way too. It's unrealistic to expect that the specifications for a program will be perfect. You're better off if you admit this up front, and write programs in a way that allows specifications to change on the fly.

この点で絵画から学ぶことができる。私はハッキングもそうあるべきだと思う。プログラムの仕様に完璧さを期待するなんて非現実的だ。そのことをまず最初に認めて、仕様がプログラムを書いている最中に変わっていても、それを受け入れられるような書き方をすべきなんだ。

(The structure of large companies makes this hard for them to do, so here is another place where startups have an advantage.)

(大企業の構造にはこれをやりにくくさせるものがあり、これもベンチャー企業に利する点だ。)

Everyone by now presumably knows about the danger of premature optimization. I think we should be just as worried about premature design-- deciding too early what a program should do.

今となっては誰もが早過ぎる最適化の危険を承知しているだろう。私は、同様に早過ぎる設計、つまりプログラムが何をすべきかを早く決め過ぎることにも気を付けるべきだと思う。

The right tools can help us avoid this danger. A good programming language should, like oil paint, make it easy to change your mind. Dynamic typing is a win here because you don't have to commit to specific data representations up front. But the key to flexibility, I think, is to make the language very [abstract](#). The easiest program to change is one that's very short.

正しい道具はこの危険を避けるのを助けてくれる。良いプログラミング言語は、油絵のように、変更を簡単にしてくれる。その点では動的な型付けの方が有利だ。最初から特定のデータ表現にコミットする必要がないからだ。しかし、柔軟性について最も重要な点は、言語をなるべく抽象的にしておくことだ。プログラムが短ければ変更もしやすい。

This sounds like a paradox, but a great painting has to be better than it has to be. For example, when Leonardo painted the portrait of [Ginevra de Benci](#) in the National Gallery, he put a juniper bush behind her head. In it he carefully painted each individual leaf. Many painters might have thought, this is just something to put in the background to frame her head. No one will look that closely at it.

もうひとつ、これは矛盾しているように聞こえるかもしれないが、偉大な絵画とは、到達すべきゴールのさらに上に到達していなければならない。例えば、レオナルド・ダ・ヴィンチがジネヴラ・ベンチの肖像画(ナショナルギャラリー所蔵)を描いた時、彼は人物の後ろに杜松の灌木を配置した。絵の中で彼は、松の葉を一枚一枚、注意深く描いた。他の多くの画家だったら、これは単に人物の背景を埋めるだけのものだと考えたかもしれない。誰もそんなにそれを注意深く見ることはしないだろうと。



GINEVRA DE BENCI
Leonardo da Vinci, c. 1475.

Not Leonardo. How hard he worked on part of a painting didn't depend at all on how closely he expected anyone to look at it. He was like Michael Jordan. Relentless.

レオナルド・ダ・ヴィンチは違った。彼にとって、絵のある部分にどれだけ手間をかけるかは、誰かがそこを見るかどうかには関係なかったのだ。彼はマイケル・ジョーダンと同じだ。妥協しないんだ。

Relentlessness wins because, in the aggregate, unseen details become visible. When people walk by the portrait of Ginevra de Benci, their attention is often immediately arrested by it, even before they look at the label and notice that it says Leonardo da Vinci. All those unseen details combine to produce something that's just stunning, like a thousand barely audible voices all singing in tune.

見えない細部も、それが組合わさることにより見えるようになる。妥協しないことはこの点で重要だ。ジネヴラ・ベンチの肖像画の横を通りかかった人々は、すぐにその絵に心を奪われる。絵のラベルを見てそれがレオナルド・ダ・ヴィンチによるものだと知るより前からだ。全ての見えない細部が組合わさることにより、まるでほとんど聞こえないかぼそい声が幾千も合わさってひとつの旋律を歌っているかのように、ある種圧倒される何かが生まれる。

Great software, likewise, requires a fanatical devotion to beauty. If you look inside good software, you find that parts no one is ever supposed to see are beautiful too. I'm not claiming I write great software, but I know that when it comes to code I behave in a way that would make me eligible for prescription drugs if I approached everyday life the same way. It drives me crazy to see code that's badly indented, or that uses ugly variable names.

偉大なソフトウェアも、同じように、美に対する熱狂的な没頭を必要とする。良いソフトウェアの中身を見てみれば、誰も見ないような箇所さえ美しく創られていることが分かるだろう。私は自分が偉大なソフトウェアを書いているなんていうつもりはないが、少なくともコードを書く時には、それと同じ調子で日常生活を送ったら医者から薬を支給されるだろうな、というような調子で書いている。めちゃくちゃにインデントされたコードとか、ひどい変数名を見ると気が狂いそうになる。

If a hacker were a mere implementor, turning a spec into code, then he could just work his way through it from one end to the other like someone digging a ditch.

But if the hacker is a creator, we have to take inspiration into account.

ハッカーが単なる実装者で、仕様をコードに置き換えているだけなら、溝を掘る作業者みたいに端から端まで順番に仕上げてゆくだろう。でもハッカーが創造者ならば、ひらめきを考えに入れなければならない。

In hacking, like painting, work comes in cycles. Sometimes you get excited about some new project and you want to work sixteen hours a day on it. Other times nothing seems interesting.

ハッキングには、絵を描く時と同じように、周期がある。ある時は新しいプロジェクトに夢中になって、1日16時間それをやり続けられる。別の時には何にも興味を持ってない。

To do good work you have to take these cycles into account, because they're affected by how you react to them. When you're driving a car with a manual transmission on a hill, you have to back off the clutch sometimes to avoid stalling. Backing off can likewise prevent ambition from stalling. In both painting and hacking there are some tasks that are terrifyingly ambitious, and others that are comfortingly routine. It's a good idea to save some easy tasks for moments when you would otherwise stall.

良い仕事をなすには、この周期を勘定に入れておかなくてはならない。そうすることによって、周期にどう対応すればよいかが分かるからだ。マニュアル車で坂を登る時は、時々クラッチを戻してやらないとエンストしてしまう。同じように、時々引いてみることは、熱意が止まってしまうのを防ぐのに良い方法だ。絵画にもハッキングにも、恐ろしいほど無謀な試みもあれば、楽にこなせる作業もある。エンストしてしまいそうな時のために、楽な作業を少し取っておくことは良いアイディアだろう。

In hacking, this can literally mean saving up bugs. I like debugging: it's the one time that hacking is as straightforward as people think it is. You have a totally constrained problem, and all you have to do is solve it. Your program is supposed to do x. Instead it does y. Where does it go wrong? You know you're going to win in the end. It's as relaxing as painting a wall.

ハッキングの場合は、バグを取っておくのもいい。私はデバッグが好きだ。デバッグは、普通の人がハッキングと聞いて連想する作業そのものだ。完全に制約された問題があり、やるべきはそれを解くことだけ。プログラムはxをするはずなのに、yをしている。どこでおかしくなっているんだろう。自分が最終的に勝利を収めるのは分かっている。これはまるで、壁を塗る時くらいに気楽なことだ。

The example of painting can teach us not only how to manage our own work, but how to work together. A lot of the great art of the past is the work of multiple hands, though there may only be one name on the wall next to it in the museum. Leonardo was an apprentice in the workshop of Verrocchio and painted one of the angels in his [Baptism of Christ](#). This sort of thing was the rule, not the exception. Michelangelo was considered especially dedicated for insisting on painting all the figures on the ceiling of the Sistine Chapel himself.

絵画の例は、自分の仕事をどう管理したらよいかを教えてくれるだけではない。どうやって他の人と一緒に仕事をしたらよいかも教えてくれる。過去の偉大な芸術作品の多くは、たとえ美術館の展示には一人しか名前を挙げられていないとしても、実際は複数の人間によって創られたものだ。レオナルド・ダ・ヴィンチはヴェロッキオの工房の見習いだったことがあり、ヴェロッキオの『キリストの洗礼』の中の天使の一人を描いた。こういったことは例外ではなく、むしろ慣例と見なされていた。ミケランジェロがシスティーナ礼拝堂の天井画の全ての像を自分で描くと主張した時、人々は彼のあまりの熱意に驚きさえしたのだ。



BAPTISM OF CHRIST

Andrea del Verrocchio, 1472-75

Leonardo is thought to have painted the angel on the far left.

As far as I know, when painters worked together on a painting, they never worked on the same parts. It was common for the master to paint the principal figures and for assistants to paint the others and the background. But you never had one guy painting over the work of another.

私の知る限り、複数の画家が一枚の絵に取り組む時、決して 2 人以上が同じ箇所を描くことはない。親方が主要人物を描き、助手が他の人物と背景を描くというのはよく行われていた。しかし、ある画家が別の画家の描いた上に描き足すということは決してなかった。

I think this is the right model for collaboration in software too. Don't push it too far. When a piece of code is being hacked by three or four different people, no one of

whom really owns it, it will end up being like a common-room. It will tend to feel bleak and abandoned, and accumulate cruft. The right way to collaborate, I think, is to divide projects into sharply defined modules, each with a definite owner, and with interfaces between them that are as carefully designed and, if possible, as articulated as programming languages.

ソフトウェアにおける協調開発でも、これは正しいモデルだと思う。協調ということあまり押し出しすぎないことだ。あるコードが、特定の所有者を決めずに 3 人も 4 人も人間にハックされると、それは共有部屋のようになる。雑然としていて見捨てられたような雰囲気が漂い、埃が積もってゆく。私が思うに、協調開発の正しい方法は、プロジェクトをはっきりと定義されたモジュールに分割し、各モジュールの所有者を決め、そして注意深く、できればプログラミング言語そのもののほどに明快に設計されたインタフェースをモジュール間に定めることだ。

Like painting, most software is intended for a human audience. And so hackers, like painters, must have empathy to do really great work. You have to be able to see things from the user's point of view.

絵画と同様、ソフトウェアも多くは人間が見て、使うものだ。だからハッカーも、画家と同じように、本当に偉大な仕事を成し遂げるには、共感する力が必要だ。ユーザの視点からものを見られるようにならなくちゃいけない。

When I was a kid I was always being told to look at things from someone else's point of view. What this always meant in practice was to do what someone else wanted, instead of what I wanted. This of course gave empathy a bad name, and I made a point of not cultivating it.

私は子供のころ、いつも、人の身になってものを考えなさいと教えられた。実際にはそう言われる時はいつでも、自分のしたいことじゃなくて他人の望むことをしなさい、という意味だった。だから共感なんてつまらないものだと思って、私はそれを磨こうとはしなかった。

Boy, was I wrong. It turns out that looking at things from other people's point of view is practically the secret of success. It doesn't necessarily mean being self-sacrificing. Far from it. Understanding how someone else sees things doesn't imply that you'll act in his interest; in some situations-- in war, for example-- you want to do exactly the opposite. [\[4\]](#)

だが、なんてこった。私は間違っていたんだ。他人の身になってものを見るというのは、本当は成功の秘密だったんだ。それは自己犠牲を意味するとは限らない。他の人のものの見方を理解したからって、その人の利益のために行動しなくちゃならないとは限らないんだ。特定の状況では、例えば戦争をしている時は、全く逆の行動をしたいと思うだろう。[4]

Most makers make things for a human audience. And to engage an audience you have to understand what they need. Nearly all the greatest paintings are paintings of people, for example, because people are what people are interested in.

多くのもの創りたちは、人間に観られ、受け取ってもらえるものを創る。観客を惹きつけるには、観客が何を必要としているかを理解しなくちゃならない。偉大な絵画作品はほとんど全て人物を描いているが、それは人物こそ、人々が興味を持つものだからだ。

Empathy is probably the single most important difference between a good hacker and a great one. Some hackers are quite smart, but when it comes to empathy are practically solipsists. It's hard for such people to design great software [5], because they can't see things from the user's point of view.

共感能力は、おそらく良いハッカーと偉大なハッカーの、たったひとつの最も重要な違いだろう。ハッカーのなかには非常に賢いが、共感するということにかけては全く自己中心主義の人々がいる。たぶんそういう人が偉大なソフトウェアをデザインするのは難しいだろう。[5] ユーザの視点でものを観ることができないからだ。

One way to tell how good people are at empathy is to watch them explain a technical question to someone without a technical background. We probably all know people who, though otherwise smart, are just comically bad at this. If someone asks them at a dinner party what a programming language is, they'll say something like "Oh, a high-level language is what the compiler uses as input to generate object code." High-level language? Compiler? Object code? Someone who doesn't know what a programming language is obviously doesn't know what these things are, either.

共感能力の良さをみるひとつの方法は、その人が技術的知識のない誰かに技術的な問題を説明する様子を観察することだ。たぶん、他の点では優れているのに、そういう説明になると滑稽なくらいへたくそな人を、誰でも知っているんじゃないか。ディナーパーティで誰かにプログラミング言語って何なの、と尋ねられると、そういう人は例えばこんなふうに言うだろう。「高級言語とは、コンパイラがそれを読んでオブジェクトコードを生成するようなものさ。」と。高級言語?コンパイラ?オブジェクトコード?プログラミング言語とは何かを知らない人が、こういう用語を知っているわけがないじゃないか。

Part of what software has to do is explain itself. So to write good software you have to understand how little users understand. They're going to walk up to the software with no preparation, and it had better do what they guess it will, because they're not going to read the manual. The best system I've ever seen in this respect was the original Macintosh, in 1985. It did what software almost never does: it just worked. [6]

ソフトウェアは、それを見ればすぐに使い方が分かるものであるべきだ。だから、良いソフトウェアを書くには、ユーザがどれだけ何も知らないかということを理解する必要がある。ユーザは何の準備もなくやって来て、いきなりソフトウェアに向かい、マニュアルなんか読もうともしないだろう。ソフトウェアはそういう人が期待するように振る舞うべきだ。この点で、私が知っている最高のシステムは、1985年当時の、オリジナルの Macintosh だ。それは他のほとんどのソフトウェアが決してなし得なかったことをなした。何も準備なしで使えたのだ。[6]

Source code, too, should explain itself. If I could get people to remember just one quote about programming, it would be the one at the beginning of *Structure and Interpretation of Computer Programs*.

ソースコードもまた、一目でその内容が分かるものであるべきだ。プログラミングに関して皆に一つだけ引用文句を覚えてもらえるならば、私は「計算機プログラムの構造と解釈」の冒頭のこの文を選ぶ。

Programs should be written for people to read, and only incidentally for machines to execute.

プログラムは、人々がそれを読むために書かれるべきである。たまたま、それが計算機で実行できるにすぎない。

You need to have empathy not just for your users, but for your readers. It's in your interest, because you'll be one of them. Many a hacker has written a program only to find on returning to it six months later that he has no idea how it works. I know several people who've sworn off Perl after such experiences. [7]

ユーザに対する共感だけでなく、コードを読む人に対する共感も必要だ。それはあなた自身のためでもある。あなた自身もあなたのコードの読者だからだ。半年前に自分の書いたプログラムを見て、それが何をするのか全く分からなかったという経験をした人も多いだろう。何人か、そういう経験を経て脱 Perl 宣言をした人を私は知っている。[7]

Lack of empathy is associated with intelligence, to the point that there is even something of a fashion for it in some places. But I don't think there's any correlation. You can do well in math and the natural sciences without having to learn empathy, and people in these fields tend to be smart, so the two qualities have come to be associated. But there are plenty of dumb people who are bad at empathy too. Just listen to the people who call in with questions on talk shows. They ask whatever it is they're asking in such a roundabout way that the hosts often have to rephrase the question for them.

共感能力の欠如は知性と関連付けられることがある。時にはそう振舞うことがひとつの流行とされることまである。でも、私は共感能力の欠如と知性の間には何の相関もないと思う。共感能力がなくても数学や自然科学で高い能力を発揮することはできるし、そういう分野にいる人は一般に聡明だから、知性が共感能力の欠如と関係あるかのように思われるのだろう。実際は、賢くもなく、共感するのもうまくない人だってたくさんいる。トークショーに電話をかけて質問してくる人々の会話を聞くだけでいい。彼らは話をあっちこっちに飛ばして言いたいことを言うから、ホストがわざわざそういう人のために質問を言い直してやらなくちゃならないことが往々にしてある。

So, if hacking works like painting and writing, is it as cool? After all, you only get one life. You might as well spend it working on something great.

さて、ハッキングが絵を描くことや小説を書くことと同じだとして、それは同じくらいクールだろうか。結局のところ、あなたには 1 回の人生しか与えられていない。他のすごいことに人生を使った方がいいかもしれないじゃないか。

Unfortunately, the question is hard to answer. There is always a big time lag in prestige. It's like light from a distant star. Painting has prestige now because of great work people did five hundred years ago. At the time, no one thought these paintings were as important as we do today. It would have seemed very odd to people at the time that Federico da Montefeltro, the Duke of Urbino, would one day be known mostly as the guy with the strange nose in a [painting](#) by Piero della Francesca.

残念ながら、この問題に答えるのは難しい。名声は大きく遅れてくるのが常だ。まるで遠くの星から届く光のように。絵画は現在では大きな価値があると認められているが、それは 500 年前に偉大な画家たちが活躍したからだ。彼らが活躍していた当時は、誰もそれを、今私たちが考えるほど価値のあるものとは思っていなかった。当時の人々は、ウルビー

ノのフェデリーコ・ダ・モンテフェルトロ公が、ピエロ・デッラ・フランチェスカの描いた絵の奇妙な鼻を持つ男、として知られる日が来ると聞いたら、ずいぶん変に思ったに違いない。



FEDERICO DA MONTEFELTRO
Piero della Francesca, 1465-66.

So while I admit that hacking doesn't seem as cool as painting now, we should remember that painting itself didn't seem as cool in its glory days as it does now. だから、ハッキングが現在、絵を描くことほどクールには思えないとしても、絵画の栄光の時代に絵を描くことは今ほどクールに思われていなかったということを気に留めておく

必要があるだろう。

What we can say with some confidence is that these are the glory days of hacking. In most fields the great work is done early on. The paintings made between 1430 and 1500 are still unsurpassed. Shakespeare appeared just as professional theater was being born, and pushed the medium so far that every playwright since has had to live in his shadow. Albrecht Durer did the same thing with engraving, and Jane Austen with the novel.

いくばくかの自信を持って言えることは、いつか、ハッキングの栄光の時代が来るだろうということだ。多くの分野で、偉大な仕事は、早い時期になされた。1430年から1500年の間に描かれた絵画はいまだに他の追従を許さない。シェークスピアは、職業としての芝居が生まれつつある時期に現れ、その表現方法をあまりに深くまで追求したために、その後の脚本家は全て、彼の影の中に生きることを強いられてきた。アルブレヒト・デューラーは版画で、ジェーン・オーステンは小説で同じことをした。

Over and over we see the same pattern. A new medium appears, and people are so excited about it that they explore most of its possibilities in the first couple generations. Hacking seems to be in this phase now.

繰り返し繰り返し、同じパターンが見られる。新しい表現方法が現れ、それに熱狂した人々が、最初の2世代くらいの間にその表現方法の可能性のほとんどを探索し尽くしてしまう。ハッキングはまさにその段階にある。

Painting was not, in Leonardo's time, as cool as his work helped make it. How cool hacking turns out to be will depend on what we can do with this new medium.

レオナルド・ダ・ヴィンチの時代には、絵画はそれほどクールではなかったが、彼の作品のおかげで後生ずっと重要視されるようになった。ハッキングがどれだけクールになるかは、私たちがこの新しい表現方法で何ができるかにかかっているんだ。

Notes

注釈

[1] The greatest damage that photography has done to painting may be the fact that it killed the best day job. Most of the great painters in history supported themselves by painting portraits.

[1] 写真が絵画に及ぼした最悪の影響は、画家たちの、一番割の良い仕事をなくしてしまったことかもしれない。歴史の中の偉大な画家の多くは肖像画を描くことで生計を立てて

いた。

[2] I've been told that Microsoft discourages employees from contributing to open-source projects, even in their spare time. But so many of the best hackers work on open-source projects now that the main effect of this policy may be to ensure that they won't be able to hire any first-rate programmers.

[2] マイクロソフトは、従業員がたとえ余暇を使っているとしてもオープンソースプロジェクトへ貢献するのをよしとしていない、という話を聞いたことがある。だが、これほどたくさん腕利きハッカーたちがオープンソースプロジェクトに参加するようになった今となつては、この方針は一流のプログラマを雇うのを難しくするだけだろう。

[3] What you learn about programming in college is much like what you learn about books or clothes or dating: what bad taste you had in high school.

[3] 大学でプログラミングについて学ぶということは、読書や服装やデートについて学ぶことに似ている。高校時代を振り返ってみたまえ。なんてセンスがなかったんだろう、って思わないか。

[4] Here's an example of applied empathy. At Viaweb, if we couldn't decide between two alternatives, we'd ask, what would our competitors hate most? At one point a competitor added a feature to their software that was basically useless, but since it was one of few they had that we didn't, they made much of it in the trade press. We could have tried to explain that the feature was useless, but we decided it would annoy our competitor more if we just implemented it ourselves, so we hacked together our own version that afternoon.

[4] 共感の応用例をひとつ挙げておこう。Viaweb で、2つの選択肢のどちらを選ぶか迷ったとき、私たちはいつも、競争相手にとってはどっちがより嫌だろうかを考えた。ある時、ある競争相手が、あまり役に立ちそうもない機能を彼らのソフトに付け足したことがあった。ただ、その機能は彼らにあって私たちにはない数少ないものだったから、彼らはマスコミ相手にそれを盛んに宣伝した。私たちはその機能が無用の長物だということを訴えかけることもできたが、それよりむしろ、うちにも同じ機能を付けたほうが相手にダメージを与えられるだろうと考え、その日の午後のうちに新しい機能を追加したんだ。

[5] Except text editors and compilers. Hackers don't need empathy to design these, because they are themselves typical users.

[5] テキストエディタとコンパイラは例外だ。ハッカーはこれらを書くのに共感能力を必要としない。典型的なユーザは自分自身だからだ。

[6] Well, almost. They overshot the available RAM somewhat, causing much inconvenient disk swapping, but this could be fixed within a few months by buying an additional disk drive.

[6] まあ、ほとんどは。時々 RAM を無駄使いして、ディスクがスワップしつづけて不便な思いをした。数ヵ月我慢して新しいディスクドライブを買えば解消できたが。

[7] The way to make programs easy to read is not to stuff them with comments. I would take Abelson and Sussman's quote a step further. Programming languages should be designed to express algorithms, and only incidentally to tell computers how to execute them. A good programming language ought to be better for explaining software than English. You should only need comments when there is some kind of kludge you need to warn readers about, just as on a road there are only arrows on parts with unexpectedly sharp curves.

[7] プログラムを読みやすくする方法は、コメントを詰め込まないようにすることだ。アベルソンとサスマンの言葉をさらに進めたい。プログラミング言語はアルゴリズムを表現するために設計されるべきで、それがたまたま、計算機で実行できるにすぎない。良いプログラミング言語は英語よりもうまくソフトウェアを説明することができるべきだ。コメントは、読者に特に気を付けてもらいたいある種の障害物を警告するためだけに必要はない。ちょうど、道路で突然の急カーブを警告するためだけに矢印の標識が用いられるように。

Thanks to Trevor Blackwell, Robert Morris, Dan Giffin, and Lisa Randall for reading drafts of this, and to Henry Leitner and Larry Finkelstein for inviting me to speak.

Chapter 3 WHAT YOU CAN'T SAY

第3章 口にできないこと

January 2004



Have you ever seen an old photo of yourself and been embarrassed at the way you looked? *Did we actually dress like that?* We did. And we had no idea how silly we looked. It's the nature of fashion to be invisible, in the same way the movement of the earth is invisible to all of us riding on it.

昔の自分の写真を見て、その格好に恥ずかしさを覚えたことがないだろうか。ほんとにこんな服を着てたんだっけ? そう、着てたんだ。それがどんなに滑稽に見えるか、その時は思いもしなかった。流行は本質的に、見えないものだ。ちょうど地球の動きが、その上に乗っている私たちにとって見えないものであるように。

What scares me is that there are moral fashions too. They're just as arbitrary, and just as invisible to most people. But they're much more dangerous. Fashion is mistaken for good design; moral fashion is mistaken for good. Dressing oddly gets you laughed at. Violating moral fashions can get you fired, ostracized, imprisoned, or even killed.

だが、道徳にも流行があることを思うと、ちょっと恐ろしくなる。道徳の流行も同じように何でもありで、しかもたいていの人には見えない。だがそれは服装の流行よりずっと危険だ。流行の服装は良いデザインと誤解されるのに対し、流行の道徳は善と誤解される。変な格好をしていても笑われるだけだが、道徳の流行に逆らうと、首になったり、仲間外れになったり、投獄されたり、殺されることさえある。

If you could travel back in a time machine, one thing would be true no matter where you went: you'd have to watch what you said. Opinions we consider harmless could have gotten you in big trouble. I've already said at least one thing that would have gotten me in big trouble in most of Europe in the seventeenth century, and did get Galileo in big trouble when he said it-- that the earth moves.

[\[1\]](#)

タイムマシンで過去に時間旅行に行けることになったとしよう。その場合、どこへ行くかにかかわらず、忠告しておきたいことがある。口にすることに気を付けよう。私たちが無

害だと思っている意見でも、大変な騒動に巻き込まれる可能性がある。私は既に、17 世紀のヨーロッパにいたなら大きな災難に遭っていただろう発言を少なくともひとつしている。実際、それを口にしたせいでガリレイは大変な目にあったんだ。つまり、地球は動く、とね。[1]

It seems to be a constant throughout history: In every period, people believed things that were just ridiculous, and believed them so strongly that you would have gotten in terrible trouble for saying otherwise.

これは、歴史のどの時点でも変わらないことに思える。どの時代においても、本当に馬鹿げた考えが信じられていることがある。しかも、反対の考えを言おうものなら厄介事に巻き込まれかねないほど強く人々に信じられていたりするんだ。

Is our time any different? To anyone who has read any amount of history, the answer is almost certainly no. It would be a remarkable coincidence if ours were the first era to get everything just right.

現代は違うのだろうか。多少でも歴史をかじったことのある人なら、まず間違いなく、現代も変わらないと答えるだろう。もし現代が、史上初めて、全てが正しく行われている時代であるとしたなら、その時代に居合わせている私たちはなんと素晴らしい幸運に恵まれていることだろうか。

It's tantalizing to think we believe things that people in the future will find ridiculous. What *would* someone coming back to visit us in a time machine have to be careful not to say? That's what I want to study here. But I want to do more than just shock everyone with the heresy du jour. I want to find general recipes for discovering what you can't say, in any era.

将来の人々が馬鹿げたことだと思うことを私たちが信じているかもしれない。そう思うと、何やら心穏やかでなくなる。もし誰かが、未来からタイムマシンに乗って現代にやってきたとして、彼が口にしないように気を付けなければならないことって何だろうか。これが、私がここで考えようとしているトピックだ。でも、ただ現代の異端を見せてみんなをびっくりさせようってだけじゃない。どんな時代においても使える、口にできないことを見つける一般的な方法を探してみるんだ。

The Conformist Test

順応性検査

Let's start with a test: Do you have any opinions that you would be reluctant to

express in front of a group of your peers?

まず、このテストから始めよう。あなたは、自分と同格の人が集まっているグループ内で、表明するのを憚るような意見を持っているだろうか。

If the answer is no, you might want to stop and think about that. If everything you believe is something you're supposed to believe, could that possibly be a coincidence? Odds are it isn't. Odds are you just think whatever you're told.

答えが「いいえ」ならば、ちょっと止まって考えてみてほしい。あなたが信じていることが全て、あなたが信じるべきとされていることだとしたら、それは偶然だろうか。確率的には、そうじゃない可能性の方が高い。人に言われたことを何であれそのまま信じている、というほうがありそうだ。

The other alternative would be that you independently considered every question and came up with the exact same answers that are now considered acceptable. That seems unlikely, because you'd also have to make the same mistakes. Mapmakers deliberately put slight mistakes in their maps so they can tell when someone copies them. If another map has the same mistake, that's very convincing evidence.

もちろん、あなたはそれぞれの疑問に対して独自の答えを考えたのだけれど、それがたまたま、現代において良いとされているものと一致した、ということもあるだろう。でも、その確率は低そうだ。完全に一致するためには、同じ間違いをしなくちゃならないからだ。地図会社はわざと小さな間違いを製品に入れておいて、誰かがコピーしたらそれと分かるようにしている。もし他社の地図にも同じ間違いがあったら、それは無断複製の有力な証拠だ。

Like every other era in history, our moral map almost certainly contains a few mistakes. And anyone who makes the same mistakes probably didn't do it by accident. It would be like someone claiming they had independently decided in 1972 that bell-bottom jeans were a good idea.

歴史上の全ての他の時代と同じように、現代の道徳もいくつか誤りがあることはまず間違いない。そして、誰かが同じ間違いをしたとしたら、おそらくそれは偶然じゃない。それを偶然だと主張するのは、自分は 1972 年にベルボトムのジーンズは格好良いと思っていたが、それは流行とは全く関係なく自分で決めたんだ、と主張するようなものだ。

If you believe everything you're supposed to now, how can you be sure you wouldn't also have believed everything you were supposed to if you had grown up

among the plantation owners of the pre-Civil War South, or in Germany in the 1930s-- or among the Mongols in 1200, for that matter? Odds are you would have.

あなたが今、信じるべきとされているものを信じているのだとしたら、もしあなたが南北戦争前の南部の農園主の家や、1930年代のドイツや、あるいは1200年のモンゴルで育っていれば、やっぱりその時代、時代に信じるべきとされているものを信じていたはずだ。

Back in the era of terms like "well-adjusted," the idea seemed to be that there was something wrong with you if you thought things you didn't dare say out loud. This seems backward. Almost certainly, there is something wrong with you if you *don't* think things you don't dare say out loud.

社会への適応が尊重されていた時代には、口にできないような考えを抱くというだけで、どこかおかしいと思われていた。だが本来は逆だ。口にできない考えを抱かないほうが、ほぼ間違いなくどこかおかしいんだ。

Trouble

災難

What can't we say? One way to find these ideas is simply to look at things people do say, and get in trouble for. [\[2\]](#)

口にできない考えには、どんなものがあるだろうか。そのような考えを見つけるひとつの方法は、他の人がそれを口にしたがために災難に巻き込まれたものごとに注目することだ。

[\[2\]](#)

Of course, we're not just looking for things we can't say. We're looking for things we can't say that are true, or at least have enough chance of being true that the question should remain open. But many of the things people get in trouble for saying probably do make it over this second, lower threshold. No one gets in trouble for saying that $2 + 2$ is 5, or that people in Pittsburgh are ten feet tall. Such obviously false statements might be treated as jokes, or at worst as evidence of insanity, but they are not likely to make anyone mad. The statements that make people mad are the ones they worry might be believed. I suspect the statements that make people maddest are those they worry might be true.

もちろん、私たちがやろうとしているのは、ただ口にできないことを探すというのではない。真実であるか、あるいは少なくとも真実である可能性が十分にあり、問題を公にしておいたほうが良いにもかかわらず、口にできないということを探しているんだ。もっとも、

口にしがために騒ぎに巻き込まれるようなことの多くは、真実である可能性はそれなりにあるだろうと思われる。2 + 2 は 5 だとか、ピッツバーグの住人は身長が 3m あるとか主張したって、災難に巻き込まれるようなことはない。そういったあからさまに間違っている主張は、ジョークや頭がおかしい証拠だと受け止められるくらいで、誰かがそれを聞いてかんかんになるってことはあまりありそうにない。人がそれを聞いて怒り出すような主張とは、その主張が信じられると困るものだ。おそらく最も人の怒りを買う主張は、人々が真実であるかもしれないと恐れている主張だろう。

If Galileo had said that people in Padua were ten feet tall, he would have been regarded as a harmless eccentric. Saying the earth orbited the sun was another matter. The church knew this would set people thinking.

ガリレイが、パドヴァの人々は身長が 3m あると言っていれば、害のない奇人だと思われるであろう。だが地球が太陽の周りを回っていると言うことは別問題だ。それによって人々がものを考え始めるようになることを、教会は知っていた。

Certainly, as we look back on the past, this rule of thumb works well. A lot of the statements people got in trouble for seem harmless now. So it's likely that visitors from the future would agree with at least some of the statements that get people in trouble today. Do we have no Galileos? Not likely.

過去を振り返ってみれば、この基本原則は確かにうまく使えるようだ。それを言ったことによって災難に巻き込まれたような意見の多くは、現在では害のないものだ。だから、未来からの旅行者は、今日人々が災難に巻き込まれているような意見の少なくともいくつかは、同じように無害なものだと考えるだろう。現代にガリレイはいないのだろうか。そんなことはあるまい。

To find them, keep track of opinions that get people in trouble, and start asking, could this be true? Ok, it may be heretical (or whatever modern equivalent), but might it also be true?

そういったケースを探すには、人々が災難に巻き込まれるような主張に常に注目して、「その主張は真実であり得るだろうか」と問うてみることだ。そういう主張は「異端」(あるいは現代でそれに相当するもの)ではあるだろうが、同時にそれが真実である可能性はないだろうか。

Heresy

異端

This won't get us all the answers, though. What if no one happens to have gotten in trouble for a particular idea yet? What if some idea would be so radioactively controversial that no one would dare express it in public? How can we find these too?

ただ、この方法で全ての答えが得られるわけではない。ある考えを口にはしたが、まだ誰も災難に巻き込まれていないとしたら? また、その主張があまりに強烈に物議を醸しそうなために、誰もそれを敢えて公に言い出せないでいるとしたら? どうやってそういう考えを見つけたらよいだろう。

Another approach is to follow that word, heresy. In every period of history, there seem to have been labels that got applied to statements to shoot them down before anyone had a chance to ask if they were true or not. "Blasphemy", "sacrilege", and "heresy" were such labels for a good part of western history, as in more recent times "indecent", "improper", and "unamerican" have been. By now these labels have lost their sting. They always do. By now they're mostly used ironically. But in their time, they had real force.

もうひとつのアプローチは、「異端」という言葉を追うことだ。歴史のどの時代でも、真偽の議論に登る前に主張を撃ち落とすために使われるレッテルがあった。西洋の歴史では長らく、「不敬」、「冒瀆」、「異端」といった言葉が使われてきたし、少し前は「不作法」、「不道德」、「アメリカ的でない」といった言葉がそうであった。しかし、そういったレッテルは既に効き目を失った。レッテルというものは常にそうだ。今では、こういった言葉は普通は皮肉としてしか使われない。だが一時期、これらの言葉は本物の力を持っていたんだ。

The word "defeatist", for example, has no particular political connotations now. But in Germany in 1917 it was a weapon, used by Ludendorff in a purge of those who favored a negotiated peace. At the start of World War II it was used extensively by Churchill and his supporters to silence their opponents. In 1940, any argument against Churchill's aggressive policy was "defeatist". Was it right or wrong? Ideally, no one got far enough to ask that.

例えば「敗北主義者」という言葉は、現代では特に政治色を帯びてはいない。しかし 1917 年のドイツにおいてはこの言葉は武器であった。ルーデンドルフはこの言葉を使って、和平案を支持する人々を一掃したのだ。第二次世界大戦の始まりにおいては、この言葉はチャーチルと支持者たちによって、反対者を黙らせるために使われた。1940 年には、チャーチルの攻撃的な政策に反対するいかなる主張も「敗北主義」とされた。このレッテルは理想的に機能した。それが正しいことか間違ったことか、そう問うところまでは誰も行かなかったんだ。

We have such labels today, of course, quite a lot of them, from the all-purpose "inappropriate" to the dreaded "divisive." In any period, it should be easy to figure out what such labels are, simply by looking at what people call ideas they disagree with besides untrue. When a politician says his opponent is mistaken, that's a straightforward criticism, but when he attacks a statement as "divisive" or "racially insensitive" instead of arguing that it's false, we should start paying attention.

もちろん現代にもそういうレッテルはある。しかもたくさん。あきれるほど何にでも使える「不適切な」という言葉から、恐れられている「対立主義的」まで。どんな時代でも、そういったレッテルを見つけるのは難しくない。同意出来ない意見に対して、「それは正しくない」という代わりに人々がどういう言葉を使うかを見れば良い。政治家が、相手に対して「間違っている」というなら、それは率直な批判だ。しかし政治家がある主張を、その真偽ではなく「対立主義的」だとか「差別に対して配慮を欠いている」だとか言って攻撃しているとすれば、注意を払ったほうがいい。

So another way to figure out which of our taboos future generations will laugh at is to start with the labels. Take a label-- "sexist", for example-- and try to think of some ideas that would be called that. Then for each ask, might this be true?

このように、私たちのタブーのうちどれが将来の世代から見て笑われるものであるかを知るもうひとつの方法は、レッテルから始めることだ。ひとつレッテルを選んでみよう(例えば、「性差別主義者」とかね)。次に、そのように呼ばれるアイディアをいくつか考えてみる。そして、それらのアイディアの一つひとつについて、それが真実である可能性を問うてみるんだ。

Just start listing ideas at random? Yes, because they won't really be random. The ideas that come to mind first will be the most plausible ones. They'll be things you've already noticed but didn't let yourself think.

そういうアイディアをランダムに書き出していこう。ランダムでいいのかって?いいんだ。実はランダムに書いたつもりでも、最初に意識に上ってくるアイディアが、一番ありそうなアイディアなんだ。そういうものは、既に気付いているにもかかわらず、考えることを止めてしまっているものだからだ。

In 1989 some clever researchers tracked the eye movements of radiologists as they scanned chest images for signs of lung cancer. [\[3\]](#) They found that even when the radiologists missed a cancerous lesion, their eyes had usually paused at

the site of it. Part of their brain knew there was something there; it just didn't percolate all the way up into conscious knowledge. I think many interesting heretical thoughts are already mostly formed in our minds. If we turn off our self-censorship temporarily, those will be the first to emerge.

1989年に、ある優秀な研究者たちが、放射線科医師が胸のレントゲンから肺癌の兆候を探す時の眼の動きを調べた。[\[3\]](#)すると、医師が癌性の病変を見逃した時でさえ、医師の眼はその箇所にも少しの間留まっていることが分かった。つまり、医師も頭のどこかでは、そこに何かがあるということを認識している。ただそれが、意識的な知識の部分まで届かないだけなのだ。多くの面白い異端的な考えは、私たちの意識の中にほとんど形になるところまでできているんじゃないかと私は思う。だから、自己検閲機能を一時的に解除してみれば、そういったものが最初に浮かび上がってくるんだ。

Time and Space

時間と空間

If we could look into the future it would be obvious which of our taboos they'd laugh at. We can't do that, but we can do something almost as good: we can look into the past. Another way to figure out what we're getting wrong is to look at what used to be acceptable and is now unthinkable.

未来を見ることができれば、私たちのタブーのどれが笑われるようになるかはたちどころにわかる。もちろん、それはできない相談だが、似たようなことならできる。過去を見るんだ。私たちが間違えていることを知るもうひとつの方法は、かつて受け入れられていたのに今では考えもできないことを探すことだ。

Changes between the past and the present sometimes do represent progress. In a field like physics, if we disagree with past generations it's because we're right and they're wrong. But this becomes rapidly less true as you move away from the certainty of the hard sciences. By the time you get to social questions, many changes are just fashion. The age of consent fluctuates like hemlines.

過去から現在への変化のいくつかは、進歩によるものだろう。物理学のような領域では、過去の世代と矛盾する結果が出てきたら、それは私たちが正しくて過去が間違っているということだ。でも、きっちりした科学の確実性から離れるにしたがって、そうはつきりと言えることは急速に少なくなる。社会的な問題にまで来れば、多くの変化は単なる流行だ。広く受け入れられるアイディアの変化は、スカート丈の変動と似たり寄ったりだ。

We may imagine that we are a great deal smarter and more virtuous than past

generations, but the more history you read, the less likely this seems. People in past times were much like us. Not heroes, not barbarians. Whatever their ideas were, they were ideas reasonable people could believe.

現代に生きる私たちの方が、過去の世代よりずっと聡明で高潔だと思いたいかもしれない。しかし歴史を学べば学ぶほど、そうは思えなくなってくる。過去に生きた人々は私たちとよく似ていた。英雄でもなく、野蛮人でもなく。彼らが何を考えていたにせよ、それは人々が信じるに足る考えであっただろう。

So here is another source of interesting heresies. Diff present ideas against those of various past cultures, and see what you get. [4] Some will be shocking by present standards. Ok, fine; but which might also be true?

さらにもうひとつ、面白い異端を見つける方法がある。現在のアイディアとさまざまな過去の文化におけるアイディアとの diff を取って結果を見てみるんだ。[4] 現在の基準からするとびっくりするようなものがきつといくつかあるだろう。それはそうだ。でもその場合、どちらが正しいと言えるだろうか。

You don't have to look into the past to find big differences. In our own time, different societies have wildly varying ideas of what's ok and what isn't. So you can try diffing other cultures' ideas against ours as well. (The best way to do that is to visit them.)

大きな違いを見つけるのに必ずしも過去を覗き見る必要はない。現代においても、何が良く何が駄目かと判断されるかは、社会によって大いに異なっている。だから自分の文化のアイディアと他の文化のそれとの diff を取ってみてもよいだろう。(それをする最良の方法は、そういう文化を訪ねてみることだ)。

You might find contradictory taboos. In one culture it might seem shocking to think x, while in another it was shocking not to. But I think usually the shock is on one side. In one culture x is ok, and in another it's considered shocking. My hypothesis is that the side that's shocked is most likely to be the mistaken one.

[5]

矛盾するタブーが見つかるだろう。ある文化では X を考えることは衝撃的だが、別の文化ではそれを考えないことが衝撃的とされるかもしれない。でも多くの場合、衝撃は一方の側にあるんじゃないかと私は思う。ある文化では何でもないことが、別の文化ではそれはとんでもないこととされる。私の仮説は、衝撃を受ける側こそが間違いを犯しているんじゃないかってことだ。[5]

I suspect the only taboos that are more than taboos are the ones that are universal, or nearly so. Murder for example. But any idea that's considered harmless in a significant percentage of times and places, and yet is taboo in ours, is a good candidate for something we're mistaken about.

タブー以上のタブーというのは、おそらくほとんど普遍的にタブーとされていることだけだと思う。例えば殺人だ。だが、昔からさまざまな土地で無害とされてきた考えが今ここでタブーとされているなら、何か間違いを犯しているのは自分たちではないかと疑ってかかったほうがよさそうだ。

For example, at the high water mark of political correctness in the early 1990s, Harvard distributed to its faculty and staff a brochure saying, among other things, that it was inappropriate to compliment a colleague or student's clothes. No more "nice shirt." I think this principle is rare among the world's cultures, past or present. There are probably more where it's considered especially polite to compliment someone's clothing than where it's considered improper. So odds are this is, in a mild form, an example of one of the taboos a visitor from the future would have to be careful to avoid if he happened to set his time machine for Cambridge, Massachusetts, 1992.

例えば、「政治的に正しい」言葉遣いの波が最高潮に達した 1990 年代始めのころ、ハーバードでは教授と職員に対してパンフレットが配られた。その中に、他の諸注意に混じって、同僚や学生の服装を褒めることは不適切であると書かれていた。もう「いいシャツだね」って言っちゃいけないんだってさ。こんな規則は、おそらく過去でも現在の文化の中でも、非常に珍しいものだろうと思う。人の服装を褒めることは逆に礼儀正しいことと考えられている文化のほうが多いだろう。したがって、これはおそらく、未来からの旅行者が行き先を 1992 年のマサチューセッツ州ケンブリッジに定めた場合に避けたほうがいいタブーの例になるのだろう。

Prigs

気取り屋

Of course, if they have time machines in the future they'll probably have a separate reference manual just for Cambridge. This has always been a fussy place, a town of i dotters and t crossers, where you're liable to get both your grammar and your ideas corrected in the same conversation. And that suggests another way to find taboos. Look for prigs, and see what's inside their heads.

もちろん、未来にタイムマシンがあるのなら、きっとケンブリッジに関するリファレンス

マニュアルが別に用意されていることだろう。ケンブリッジはいつでも、細かなことに口を出す小うるさい人々の街であり、会話をしている間にその内容と文法を同時に直されるのを覚悟しなくちゃならないような場所だから。ここにもうひとつ、タブーを発見する方法があるかもしれない。堅苦しく、気取った人を探して、その頭の中を覗いてみるんだ。

Kids' heads are repositories of all our taboos. It seems fitting to us that kids' ideas should be bright and clean. The picture we give them of the world is not merely simplified, to suit their developing minds, but sanitized as well, to suit our ideas of what kids ought to think. [6]

子供の頭の中には、ある意味、私たちのタブーの全てが保存されている。私たちは、子供の考えは明るくて汚れないものであるべきだと思っていたがる。私たちが子供に見せる世界の姿は、発達中の心に合わせて単純にしてあるだけでなく、きちんと消毒されている。子供はこう考えるべき、という大人の考えに合わせるために。[6]

You can see this on a small scale in the matter of dirty words. A lot of my friends are starting to have children now, and they're all trying not to use words like "fuck" and "shit" within baby's hearing, lest baby start using these words too. But these words are part of the language, and adults use them all the time. So parents are giving their kids an inaccurate idea of the language by not using them. Why do they do this? Because they don't think it's fitting that kids should use the whole language. We like children to seem innocent. [7]

汚い言葉の扱いにその縮図を見ることができる。私の友人の多くはそろそろ子供を育て始めたが、みんな赤ん坊が聞いているところでは「fuck」とか「shit」なんて言葉は使わないようにしている。そうしないと赤ん坊がそういう言葉を使い始めてしまうかのように。でも、こういう単語は言語の一部であって、大人はいつでも使っているじゃないか。ということは、大人は自分たちの子供に対して、それらの単語を使わないことで言語に関する不正確な考えを伝えていることになる。どうしてそんなことをするんだろうか。それは、子供にとってはふさわしくない言葉もあると大人が考えているからだ。子供には無垢でいてほしいのだ。[7]

Most adults, likewise, deliberately give kids a misleading view of the world. One of the most obvious examples is Santa Claus. We think it's cute for little kids to believe in Santa Claus. I myself think it's cute for little kids to believe in Santa Claus. But one wonders, do we tell them this stuff for their sake, or for ours?

たいていの大人は、同じ理由から、わざと世界に関して誤解を生みやすい見方を子供に与える。最も明白な例のひとつはサンタクロースだ。そりゃ、小さな子がサンタクロースを

信じているのは可愛いものだ。私自身も、小さな子がサンタクロースを信じているのを可愛いと思う。ただ、こういう疑問は残る。大人は、子供自身のためにそういうことを言っているのだろうか、それとも大人自身のためにだろうか。

I'm not arguing for or against this idea here. It is probably inevitable that parents should want to dress up their kids' minds in cute little baby outfits. I'll probably do it myself. The important thing for our purposes is that, as a result, a well brought-up teenage kid's brain is a more or less complete collection of all our taboos-- and in mint condition, because they're untainted by experience. Whatever we think that will later turn out to be ridiculous, it's almost certainly inside that head.

ここでは、この考えに賛成だとか反対だとかいう議論をしているわけではない。親というものは、子供の世界を可愛い赤ちゃん向けグッズで飾り立てずにはおれないものなんだろう。私だって自分の子にはそう思う。ここでの目的にとって重要なことは、結果として、大切に育てられたティーンエイジの子供の頭の中には、私たちにとってのタブーのほぼ完璧なコレクションが作られることになる、ということだ。しかも、経験によって汚されてはいない、真新しい状態で。将来馬鹿げたことだと分かるアイディアは何であれ、既にその頭の中に入っているはずだ。

How do we get at these ideas? By the following thought experiment. Imagine a kind of latter-day Conrad character who has worked for a time as a mercenary in Africa, for a time as a doctor in Nepal, for a time as the manager of a nightclub in Miami. The specifics don't matter-- just someone who has seen a lot. Now imagine comparing what's inside this guy's head with what's inside the head of a well-behaved sixteen year old girl from the suburbs. What does he think that would shock her? He knows the world; she knows, or at least embodies, present taboos. Subtract one from the other, and the result is what we can't say.

ここからどうやって口にできないアイディアを取り出すかって?次の思考実験を使うんだ。後年のコンラッドの小説に出てくるような男を想定してみよう。彼はある時はアフリカで傭兵となり、ある時はネパールで医師となり、そしてまたある時はマイアミのナイトクラブで支配人を務めるなど、いや、細かいことは何でもよいのだが、とにかく彼は実にいろいろなことを見聞きしてきた。この男の頭の中と、郊外で行儀良く育てられた 16 歳の女の子の頭の中を比較してみたらどうだろう。彼が考えることのうち、女の子に衝撃を与えるようなことはいったい何だろう。彼は世界を知っている。そして女の子は、私たちのタブーを知って、いや少なくとも内包している。一方からもう一方を引き算すれば、その結果が、私たちが口にできないことだ。

Mechanism

メカニズム

I can think of one more way to figure out what we can't say: to look at how taboos are created. How do moral fashions arise, and why are they adopted? If we can understand this mechanism, we may be able to see it at work in our own time.

口にできないことを見出す方法を、もうひとつ考えることができる。タブーがどうやって作られるかを見ることだ。道徳の流行はどのように生まれ、社会に根付くのだろう。このメカニズムを理解できれば、それが今の時代に動いていることを見つけられるかもしれない。

Moral fashions don't seem to be created the way ordinary fashions are. Ordinary fashions seem to arise by accident when everyone imitates the whim of some influential person. The fashion for broad-toed shoes in late fifteenth century Europe began because Charles VIII of France had six toes on one foot. The fashion for the name Gary began when the actor Frank Cooper adopted the name of a tough mill town in Indiana. Moral fashions more often seem to be created deliberately. When there's something we can't say, it's often because some group doesn't want us to.

道徳の流行のしかたは、普通の流行のしかたとは異なる。普通の流行は、誰か影響力のある人の気まぐれな思い付きをみんなが真似することで、偶然起こるようだ。15世紀後半にヨーロッパでつま先の大きな靴が流行ったのは、フランスのシャルル8世が足指を6本持っていたからだ。フランク・クーパーがインディアナ州の工場街の名前を芸名としたことで、「ゲイリー」という名前が始まった。しかし、道徳の流行はもっと意図的に作られることが多い。口にさせないことがある場合、たいていは、何らかの集団が人々にそれを口にしておほしくないと思っているからだ。

The prohibition will be strongest when the group is nervous. The irony of Galileo's situation was that he got in trouble for repeating Copernicus's ideas. Copernicus himself didn't. In fact, Copernicus was a canon of a cathedral, and dedicated his book to the pope. But by Galileo's time the church was in the throes of the Counter-Reformation and was much more worried about unorthodox ideas.

その集団が神経を尖らせていればいるほど、禁じる圧力は強くなる。ガリレイにとって皮肉だったのは、彼はコペルニクスの考えを繰り返しただけで弾圧されたことだ。コペルニクス自身は弾圧されなかったのに。実際、コペルニクスは大聖堂の参事会員であり、教皇

に自身の本を献上してさえいる。だがガリレイの時代には教会は反宗教改革の苦闘の最中にあり、非伝統的な考えに対する憂慮がはるかに強かった。

To launch a taboo, a group has to be poised halfway between weakness and power. A confident group doesn't need taboos to protect it. It's not considered improper to make disparaging remarks about Americans, or the English. And yet a group has to be powerful enough to enforce a taboo. Coprophiles, as of this writing, don't seem to be numerous or energetic enough to have had their interests promoted to a lifestyle.

タブーを作り出す集団は、弱さと権力の間で危うく揺れている。自信のある集団は、自分を守るためのタブーを必要としない。アメリカ人やイギリス人をけなす発言をしても、不適切だとは思われないだろう。一方で、集団はタブーを強制するに足る力を持っている必要がある。この原稿を書いている時点では、汚物愛好者はその嗜好をライフスタイルとして社会に受け入れさせるほどの勢力はまだないようだ。

I suspect the biggest source of moral taboos will turn out to be power struggles in which one side only barely has the upper hand. That's where you'll find a group powerful enough to enforce taboos, but weak enough to need them.

おそらく、道徳的なタブーの最大の源は、勢力のぶつかり合いにおいて一方がかろうじて他方の上手を取っているだけ、といった場合なのではないかと思う。そこには、タブーを強制するだけの力を持ち、かつそれを必要とするほど弱い、という集団がある。

Most struggles, whatever they're really about, will be cast as struggles between competing ideas. The English Reformation was at bottom a struggle for wealth and power, but it ended up being cast as a struggle to preserve the souls of Englishmen from the corrupting influence of Rome. It's easier to get people to fight for an idea. And whichever side wins, their ideas will also be considered to have triumphed, as if God wanted to signal his agreement by selecting that side as the victor.

多くの紛争は、それが本来は何についてであったにせよ、競合する思想間の争いへと変容してゆく。イギリスの宗教改革は、底では富と権力を求める闘いであったが、結局はローマの破壊的影響から英国人の魂を守るための闘いへと変容した。思想のためなら、人々は簡単に争う。そしてどちらが勝っても、勝ったほうの思想が勝者と共に勝利を誇る。まるで神が、そちらを勝たせることによってその思想に合意することを示したとでも言わなければならない。

We often like to think of World War II as a triumph of freedom over totalitarianism. We conveniently forget that the Soviet Union was also one of the winners.

私たちはしばしば、第二次世界大戦は、全体主義に対する自由の勝利だと考えたがる。その時には、ソビエト連邦も勝ち組に入っていたことは都合良く忘れていくわけだ。

I'm not saying that struggles are never about ideas, just that they will always be made to seem to be about ideas, whether they are or not. And just as there is nothing so unfashionable as the last, discarded fashion, there is nothing so wrong as the principles of the most recently defeated opponent. Representational art is only now recovering from the approval of both Hitler and Stalin. [8]

思想に基づく争いが全くないと言っているのではない。どんな争いであれ、それが思想に基づくかどうかにかかわらず、いずれそれは思想に関する争いであるかのようになるということを指摘しているのだ。そして、廃れたばかりの流行が最も流行外れであるのと同じように、打ち負かされたばかりの思想は、最も間違っただけのものであるように受け取られる。具象主義芸術は、ヒトラーとスターリンに気に入られたことから受けた痛手から、今ようやく立ち直りつつある。[8]

Although moral fashions tend to arise from different sources than fashions in clothing, the mechanism of their adoption seems much the same. The early adopters will be driven by ambition: self-consciously cool people who want to distinguish themselves from the common herd. As the fashion becomes established they'll be joined by a second, much larger group, driven by fear. [9] This second group adopt the fashion not because they want to stand out but because they are afraid of standing out.

道徳の流行は服装の流行とは違うところから発生するが、それが社会に広まるメカニズムにはさして違いがない。まず、流行りものに敏感な層が野心からそれを取り込む。自分で自分をクールだと思っている人々は、並の集団から自分を区別したいのだ。流行が確立すると、第二の、ずっと大きな集団が、恐れからその流行に加わる。[9] それらの人々は、目立ちたいからではなく、目立つのが怖いから流行を採り入れるのだ。

So if you want to figure out what we can't say, look at the machinery of fashion and try to predict what it would make unsayable. What groups are powerful but nervous, and what ideas would they like to suppress? What ideas were tarnished by association when they ended up on the losing side of a recent struggle? If a self-consciously cool person wanted to differentiate himself from preceding fashions (e.g. from his parents), which of their ideas would he tend to reject?

What are conventional-minded people afraid of saying?

したがって、口にできないことが何かを知りたいければ、流行の仕組みをよく見て、それが何を口にできなくするかを予測すればよい。力を持ち神経を尖らせている集団はいるか。そしてその集団が抑え込みたい考えとは何か。ある集団が最近の争いに敗れ、それが支持していたがために一緒に色褪せてしまった考えは何か。自意識的でクールにありたいと思う人間が、自分を前の流行(例えば親の世代の流行など)から差別化したいと考えた時に、前の流行のどんな考えを拒否するだろうか。そして普通の人々が口にすることを恐れることとはなんだろうか。

This technique won't find us all the things we can't say. I can think of some that aren't the result of any recent struggle. Many of our taboos are rooted deep in the past. But this approach, combined with the preceding four, will turn up a good number of unthinkable ideas.

このテクニックは、口にできないこと全てを発見できるわけではない。最近の争いごとの結果生まれたものではない口にできないこと、というのはいくつか思い付く。私たちのタブーの多くは、ずっと過去に深く根を下ろしている。しかしこのアプローチを前に述べた4つと組み合わせれば、ずいぶんたくさんの「考えることも憚られる考え」を見つけることができるだろう。

Why

理由

Some would ask, why would one want to do this? Why deliberately go poking around among nasty, disreputable ideas? Why look under rocks?

なんでわざわざこんなことをしたがるのか、と疑問に思う人もいるだろう。何だって、褒められない危険な考えをわざわざ覗いて回るんだい、と。どうして石の下を覗くような真似をするんだい。

I do it, first of all, for the same reason I did look under rocks as a kid: plain curiosity. And I'm especially curious about anything that's forbidden. Let me see and decide for myself.

私がそれをするのは、まず第一には、子供の時に石の下を覗いたのと同じ理由からだ。純粋な好奇心。そして特に、禁じられていることに対する好奇心。とにかく見せてよ、それからそれをどうするか自分で考えるから。

Second, I do it because I don't like the idea of being mistaken. If, like other eras,

we believe things that will later seem ridiculous, I want to know what they are so that I, at least, can avoid believing them.

第二の理由は、私は間違いを信じていたくはないからだ。他の時代と同じように、将来馬鹿げたことだと思われるようなことを私たちが信じているのだとしたら、少なくとも私は、そんなことを信じてしまわないように、それが何であるかを知っておきたい。

Third, I do it because it's good for the brain. To do good work you need a brain that can go anywhere. And you especially need a brain that's in the habit of going where it's not supposed to.

第三に、それは頭脳にとって良いからだ。良い仕事をするにはどこへでも行ける頭脳が必要だ。とりわけ、本来行くべきでないようなところに行く習性を持っているような頭脳がね。

Great work tends to grow out of ideas that others have overlooked, and no idea is so overlooked as one that's unthinkable. Natural selection, for example. It's so simple. Why didn't anyone think of it before? Well, that is all too obvious. Darwin himself was careful to tiptoe around the implications of his theory. He wanted to spend his time thinking about biology, not arguing with people who accused him of being an atheist.

素晴らしい仕事は、他人が見過ごしていたようなアイディアから生まれることが多い。そして、考えもできないようなアイディアほど、見過ごされているものはないじゃないか。例えば自然淘汰を考えてみよう。とても単純だ。だのになぜ、他の人は考えつかなかったんだろう。それは、あまりに明白過ぎたからだ。ダーウィン本人さえも、自身の理論が意味するところを注意深く、忍び足で避けて通っている。彼は自分の時間を生物学を考えて過ごしたかったわけで、彼を無神論者と弾劾する人々との議論に費やしたかったわけじゃない。

In the sciences, especially, it's a great advantage to be able to question assumptions. The m.o. of scientists, or at least of the good ones, is precisely that: look for places where conventional wisdom is broken, and then try to pry apart the cracks and see what's underneath. That's where new theories come from.

特に科学では、当然と見なされていることに疑問を持つ能力は大きな利点となる。少なくとも良い科学者の仕事手法とはまさにそうだからからだ。一般的な知識が綻びるところを探し、割け目をこじ開けて、その下にあるものを見る。そこから新しい理論が生まれる。

A good scientist, in other words, does not merely ignore conventional wisdom, but

makes a special effort to break it. Scientists go looking for trouble. This should be the m.o. of any scholar, but scientists seem much more willing to look under rocks.

[\[10\]](#)

良い科学者とは、一般的な知識を単に無視するだけではなく、それを破るために特に努力する人たちだとも言える。科学者は、自ら困難を探すのだ。これは全ての学者にとって必要な手法であるはずだが、科学者は特に石の下を覗きたがる人種であるように思える。[\[10\]](#)

Why? It could be that the scientists are simply smarter; most physicists could, if necessary, make it through a PhD program in French literature, but few professors of French literature could make it through a PhD program in physics. Or it could be because it's clearer in the sciences whether theories are true or false, and this makes scientists bolder. (Or it could be that, because it's clearer in the sciences whether theories are true or false, you have to be smart to get jobs as a scientist, rather than just a good politician.)

なぜだろう？単に、科学者は頭脳明晰だからだろうか。多くの物理学者は、必要とあらば、フランス文学の博士課程を終えることだってできるだろうが、フランス文学の教授で物理学の博士課程を終えられる人はほとんどいないだろう。それとも、科学においてはものごとの真偽がよりはっきりとしているため、科学者は大胆になれるということだろうか。(あるいは、ものごとの真偽がよりはっきりとしているため、科学者としてやってゆくためには、単に良い政治家であること以上に、聡明であることが求められるとも言えるかもしれないが)。

Whatever the reason, there seems a clear correlation between intelligence and willingness to consider shocking ideas. This isn't just because smart people actively work to find holes in conventional thinking. I think conventions also have less hold over them to start with. You can see that in the way they dress.

理由は何であれ、知性と衝撃的なアイデアへの探求心との間にははっきりとした相関があるように思える。聡明な人々は一般的な考えの穴を探すことに熱心だというだけではない。そういう人々は、常識にとらわれることが少ないともいえる。彼らの格好を見ればわかるだろう。

It's not only in the sciences that heresy pays off. In any competitive field, you can [win big](#) by seeing things that others daren't. And in every field there are probably heresies few dare utter. Within the US car industry there is a lot of hand-wringing now about declining market share. Yet the cause is so obvious that any observant outsider could explain it in a second: they make bad cars. And they

have for so long that by now the US car brands are antibrands-- something you'd buy a car despite, not because of. Cadillac stopped being the Cadillac of cars in about 1970. And yet I suspect no one dares say this. [\[11\]](#) Otherwise these companies would have tried to fix the problem.

異端が利益をもたらすのは科学の分野だけではない。競争的な領域ならどこでも、他の人が見る勇気を持たないところを見ることによって大きく勝つことができる。そしてどんな領域にも、ほとんどの人が恐れて口にしない異端というものがあるんじゃないか。米国の自動車産業は、マーケットシェアの減少に対してどう対応すべきか実に苦慮している。だが、外側にいる注意深い人間なら誰でも、その原因はあまりに明白で一言で説明できるだろう。要するに品質が悪いんだ。さらに、米国の自動車産業はそのような車を長く生産し続けてきたため、米国の車のブランドはいまや反ブランドとなってしまった。〇〇の車だから買う、ではなく、〇〇の車だけど買う、というわけだ。キャデラックが高級車の代名詞であったのは 1970 年ごろまでだ。でも、だれもそれを敢えて口にしようとはしない。[\[11\]](#) でなければ、これらのメーカーはとっくに問題を解決しようとしているはずだ。

Training yourself to think unthinkable thoughts has advantages beyond the thoughts themselves. It's like stretching. When you stretch before running, you put your body into positions much more extreme than any it will assume during the run. If you can think things so outside the box that they'd make people's hair stand on end, you'll have no trouble with the small trips outside the box that people call innovative.

考えもできないようなことを考えようとすることは、それによって得られる考え以上の効用がある。それはちょうどストレッチ運動みたいなものだ。走る前によくストレッチしておけば、普通に走るよりもずっと違う体勢を容易に取ることができるだろう。他人が聞いたら身の毛がよだつようなあまりに常軌を逸したことを考えられるようになっていれば、常識の箱のほんの外側を散歩するくらいわけもない。そして人はそれを革新的だと言うだろう。

Pensieri Stretti

Pensieri Stretti

When you find something you can't say, what do you do with it? My advice is, don't say it. Or at least, pick your battles.

口にできないことを発見したら、それをどうしたらいいだろう。私なら、それを口にしないようにと忠告する。少なくとも、不毛な争いは避けて通るべきだ。

Suppose in the future there is a movement to ban the color yellow. Proposals to paint anything yellow are denounced as "yellowist", as is anyone suspected of liking the color. People who like orange are tolerated but viewed with suspicion. Suppose you realize there is nothing wrong with yellow. If you go around saying this, you'll be denounced as a yellowist too, and you'll find yourself having a lot of arguments with anti-yellowists. If your aim in life is to rehabilitate the color yellow, that may be what you want. But if you're mostly interested in other questions, being labelled as a yellowist will just be a distraction. Argue with idiots, and you become an idiot.

例えば、将来、黄色を禁じるような運動が起こったとしよう。何かを黄色に塗ろうとしたら、黄色を好むそぶりをみせたりしたら、「黄色主義者」というレッテルが張られるんだ。オレンジ色を好む人は一応許されるが、疑わしい目で見られる。さて、そこであなたが、黄色には何も悪いところはないじゃないか、と気付いたとする。それを言い触らして回ったら、たぶんあなたも黄色主義者と呼ばれ、反黄色主義者との際限ない議論に巻き込まれるだろう。もちろん、あなたの人生の目的が黄色の復権にあるのなら、闘いは望むところだろう。だが、あなたのやりたいのがいろいろな問題を考えることだとすれば、黄色主義者と呼ばれることは邪魔でしかない。愚か者と議論する者は愚か者になるのだ。

The most important thing is to be able to think what you want, not to say what you want. And if you feel you have to say everything you think, it may inhibit you from thinking improper thoughts. I think it's better to follow the opposite policy. Draw a sharp line between your thoughts and your speech. Inside your head, anything is allowed. Within my head I make a point of encouraging the most outrageous thoughts I can imagine. But, as in a secret society, nothing that happens within the building should be told to outsiders. The first rule of Fight Club is, you do not talk about Fight Club.

最も重要なことは、考えたいことを考えられるということであって、言いたいことを言えるということではない。考えたことを全部言わなくちゃならないと思っているなら、それは不適切な考えを抱くことをかえって邪魔してしまう。逆の方針を取ったほうが良いだろう。考えることと、話すことの間にはっきりとした線を引くんだ。頭の中では全て許される。私は自分の頭の中では、思い付く限り最もとんでもない考えを押し進めるようにしている。でも、ちょうど秘密結社みたいに、その中で起きたことは外には漏らさない。ファイトクラブの最初の規約は、ファイトクラブについてしゃべらない、ということだ。

When Milton was going to visit Italy in the 1630s, Sir Henry Wootton, who had been ambassador to Venice, told him his motto should be "*i pensieri stretti & il*

viso sciolto." Closed thoughts and an open face. Smile at everyone, and don't tell them what you're thinking. This was wise advice. Milton was an argumentative fellow, and the Inquisition was a bit restive at that time. But I think the difference between Milton's situation and ours is only a matter of degree. Every era has its heresies, and if you don't get imprisoned for them you will at least get in enough trouble that it becomes a complete distraction.

ミルトンが 1630 年代にイタリアを訪ねた時、ベニスの大使であったヘンリー・ウートン卿は、"i pensieri stretti & il viso sciolto." (閉じた考えと開いた顔) をモットーにすべし、と言った。皆に微笑みかけ、しかし何を考えているかは口にしないこと。これは賢明なアドバイスであった。ミルトンは議論好きな質だったし、当時は異端審問が厳しかった。だが、ミルトンの置かれた状況と私たちの状況の差は、単に程度の差であるように私には思える。どの時代にも異端は存在し、たとえそれで投獄されることはなくても、本来の仕事をすっかり邪魔されるほどのごたごたには巻き込まれる。

I admit it seems cowardly to keep quiet. When I read about the harassment to which the Scientologists subject their critics [\[12\]](#), or that pro-Israel groups are "compiling dossiers" on those who speak out against Israeli human rights abuses [\[13\]](#), or about people being sued for violating the DMCA [\[14\]](#), part of me wants to say, "All right, you bastards, bring it on." The problem is, there are so many things you can't say. If you said them all you'd have no time left for your real work. You'd have to turn into Noam Chomsky. [\[15\]](#)

黙っていることは臆病者に見えるかもしれない。サイエントロジー信者が批判者に対して嫌がらせを行っているという記事[\[12\]](#) や、イスラエルの人権侵害に反対する人々が反ユダヤ主義者とレッテルを貼られること[\[13\]](#) や、DMCA 違反で研究者が訴訟に脅かされていること[\[14\]](#) などを読むたびに、私の中に、「もう我慢ならねえ。ぶちかましてやろうぜ」と囁くものがある。問題は、口にできないことが多過ぎることだ。全部言ってしまったら、本来の仕事にかけられる時間など無くなってしまう。ノーム・チョムスキーになるしかない。[\[15\]](#)

The trouble with keeping your thoughts secret, though, is that you lose the advantages of discussion. Talking about an idea leads to more ideas. So the optimal plan, if you can manage it, is to have a few trusted friends you can speak openly to. This is not just a way to develop ideas; it's also a good rule of thumb for choosing friends. The people you can say heretical things to without getting jumped on are also the most interesting to know.

だが、考えを秘密にしておくことには、議論の利点を失ってしまうという問題がある。ア

アイデアは話すことでもっと多くのアイデアを生み出してゆく。したがって一番良い方法は、できれば少数の何でも話せる信頼できる友人を持っておくことだ。これはアイデアを培う方法というだけでなく、友人を選ぶ良い指針にもなる。とんでもないアイデアを聞いても飛び上がらないような人こそ、知り合って面白い人たちだからだ。

Viso Sciolto?

Viso Sciolto?

I don't think we need the *viso sciolto* so much as the *pensieri stretti*. Perhaps the best policy is to make it plain that you don't agree with whatever zealotry is current in your time, but not to be too specific about what you disagree with. Zealots will try to draw you out, but you don't have to answer them. If they try to force you to treat a question on their terms by asking "are you with us or against us?" you can always just answer "neither".

viso sciolto 「開いた顔」、に関しては、pensieri stretti 「閉じた考え」ほど大事ではないんじゃないかと思う。最良の選択は、その時に熱狂している人々には与しない、とはっきり言っておき、しかしいったいどの点に同意しないかについてはぼかしておくことだろう。熱狂者たちはあなたを議論に引っ張り込もうとするだろうが、答える必要なんてないんだ。もしそういう人々が、彼らの言葉によって問題を定義して「賛成か反対か」とあなたに迫ったなら、「どちらでもない」と答えればいいだけのことだ。

Better still, answer "I haven't decided." That's what Larry Summers did when a group tried to put him in this position. Explaining himself later, he said "I don't do litmus tests." [16] A lot of the questions people get hot about are actually quite complicated. There is no prize for getting the answer quickly.

もっと良いのは、「まだ決めていない」と答えることだ。これはラリー・サマーズが、そういう立場に追い込まれそうになった時の答えだ。後に彼はその時を振り返って、「私はリトマス試験は受けない」と説明した。[16] 人々が熱くなるような問題はたいてい、とても複雑なものだ。即答できたからって何も良いことはない。

If the anti-yellowists seem to be getting out of hand and you want to fight back, there are ways to do it without getting yourself accused of being a yellowist. Like skirmishers in an ancient army, you want to avoid directly engaging the main body of the enemy's troops. Better to harass them with arrows from a distance.

反黄色主義者があまりにあなたの手に余るようになり、反撃しなければならなくなったとしても、黄色主義者として弾劾されることを避けつつ戦う方法がある。昔の軍隊の投槍兵

のように、敵の本隊との直接対決を避け、代わりに遠くから牽制するんだ。

One way to do this is to ratchet the debate up one level of abstraction. If you argue against censorship in general, you can avoid being accused of whatever heresy is contained in the book or film that someone is trying to censor. You can attack labels with meta-labels: labels that refer to the use of labels to prevent discussion. The spread of the term "political correctness" meant the beginning of the end of political correctness, because it enabled one to attack the phenomenon as a whole without being accused of any of the specific heresies it sought to suppress.

ひとつの方法は、議論の抽象度を一段上げることだ。一般的な検閲に関して議論すれば、誰かが検閲したいと思っている本や映画に含まれている異端をもとに攻撃されることを避けられる。レッテル貼りには、メタレッテル貼りで対抗するんだ。議論を避けるためのレッテル貼りそのものにレッテルを貼るわけだ。「政治的に正しい」という用語が広まったのは、いわゆる政治的に正しい言い回しを要求する風潮の終わりの始まりであった。なぜなら、政治的に正しい言い回しなるものが本当に抑圧したい特定の異端に直接触れずに、そのような風潮全体を批判することを可能にしたからだ。

Another way to counterattack is with metaphor. Arthur Miller undermined the House Un-American Activities Committee by writing a play, "The Crucible," about the Salem witch trials. He never referred directly to the committee and so gave them no way to reply. What could HUAC do, defend the Salem witch trials? And yet Miller's metaphor stuck so well that to this day the activities of the committee are often described as a "witch-hunt."

もうひとつの反撃法は、メタファーを使うことだ。アーサー・ミラーは、セイラムの魔女裁判を描いた戯曲「るつぼ」を書くことで、非米活動委員会(House Un-American Activities Committee, HUAC)を揶揄した。もちろん、彼は直接に委員会を名指しして発言することは決してなかったから、委員会もどうすることもできなかった。だって HUAC はセイラムの魔女裁判を弁護するわけにもいかなかったからね。それでも、ミラーの比喩は実にうまくはまったので、今日でも HUAC の活動はしばしば「魔女狩り」と呼ばれる。

Best of all, probably, is humor. Zealots, whatever their cause, invariably lack a sense of humor. They can't reply in kind to jokes. They're as unhappy on the territory of humor as a mounted knight on a skating rink. Victorian prudishness, for example, seems to have been defeated mainly by treating it as a joke. Likewise its reincarnation as political correctness. "I am glad that I managed to

write 'The Crucible,'" Arthur Miller wrote, "but looking back I have often wished I'd had the temperament to do an absurd comedy, which is what the situation deserved." [\[17\]](#)

最良の方法は、おそらく、ユーモアだ。熱狂している人々は、原因のいかんにかかわらず、総じてユーモアのセンスを欠いている。ジョークに対して応える術を知らない。ユーモアの領域では、まるで彼らはスケートリンクにつれてこられた重装備の騎士みたいなものだ。例えばビクトリア朝的な性的潔癖主義は、たぶんそれがジョークとして扱われることでうち負かされたのだろう。その一種の生まれ変わりである「政治的な正しさ」も同じだ。アーサー・ミラーは後に書いている。『『るつぽ』を書き通せたこと自体は嬉しく思っている。でも振り返ってみると、馬鹿げた喜劇にすればよかったと思うことが時々ある。実際、本当に馬鹿げた時代だったよ』。 [\[17\]](#)

ABQ

問い続けること

A Dutch friend says I should use Holland as an example of a tolerant society. It's true they have a long tradition of comparative open-mindedness. For centuries the low countries were the place to go to say things you couldn't say anywhere else, and this helped to make the region a center of scholarship and industry (which have been closely tied for longer than most people realize). Descartes, though claimed by the French, did much of his thinking in Holland.

オランダ人の友人は、寛容な社会の例としてオランダを出すといい、と言ってくれた。確かに、かの国は比較的心が広いことで長く知られている。何世紀にもわたって、低地諸国は、他の国では言えないようなことを言うために向かう場所であった。この地域が学問と工業の中心であったのはそのためだ(この関係は、人々が気付いているよりもずっと長期間にわたる)。デカルトはフランス人であると言われるが、多くの思索をオランダで行った。

And yet, I wonder. The Dutch seem to live their lives up to their necks in rules and regulations. There's so much you can't do there; is there really nothing you can't say?

それでも私は思うのだ。オランダ人として、日常生活では首まで規則と規制に浸かって生きているんじゃないかと。かの地でも、やったらまずいことというのは確かにたくさんある。それなのに、口にしたらまずいことはないなんてことがあるだろうか。

Certainly the fact that they value open-mindedness is no guarantee. Who thinks they're not open-minded? Our hypothetical prim miss from the suburbs thinks

she's open-minded. Hasn't she been taught to be? Ask anyone, and they'll say the same thing: they're pretty open-minded, though they draw the line at things that are really wrong. (Some tribes may avoid "wrong" as judgemental, and may instead use a more neutral sounding euphemism like "negative" or "destructive".)

心の広さを重視する、なんてせりふには何の保証もない。だいたい、心が狭いことを自認する人なんているだろうか。私たちの仮説に登場した郊外のお上品なお嬢さんだって、自分の心は広いと思っているだろう。心を広く持ちなさいって教えられて育つたろうからね。誰に聞いたって同じように答えるだろう。心は広い、けれども本当に悪いこととの間には線を引いているがね、とね。(「悪い」という言葉を審判的であるとして、もっと中立的に聞こえる、「ネガティブ」とか「破壊的」というような言葉を使う人もいるかもしれないが。)

When people are bad at math, they know it, because they get the wrong answers on tests. But when people are bad at open-mindedness they don't know it. In fact they tend to think the opposite. Remember, it's the nature of fashion to be invisible. It wouldn't work otherwise. Fashion doesn't seem like fashion to someone in the grip of it. It just seems like the right thing to do. It's only by looking from a distance that we see oscillations in people's idea of the right thing to do, and can identify them as fashions.

数学が出来ない時、人々は自分でそれがわかる。テストの答えが間違っているからだ。でも、心の広さに関しては、自分では分からないんだ。むしろ、心の狭い人のほうが自分の心は広いと考えがちだ。流行の本質は見えなくなることだってことを思い出してほしい。そうでないと流行にならないからだ。流行にどっぷりと染まった人にとっては、流行は流行には思えない。ただ、それをするのが正しいことのように思えるだけだ。距離を置いて眺めてみて初めて、私たちは人々が正しいと考えることの揺らぎを見ることができ、それが流行であると識別できるのだ。

Time gives us such distance for free. Indeed, the arrival of new fashions makes old fashions easy to see, because they seem so ridiculous by contrast. From one end of a pendulum's swing, the other end seems especially far away.

時間はそのような距離をただで与えてくれる。実際、新しい流行の到来によって、古い流行は見るのが容易になる。対比してみればずっと馬鹿げて見えるからね。振り子の一方の端からは、他方の端ははるか遠くに見えるものだ。

To see fashion in your own time, though, requires a conscious effort. Without time to give you distance, you have to create distance yourself. Instead of being part of the mob, stand as far away from it as you can and watch what it's doing. And pay

especially close attention whenever an idea is being suppressed. Web filters for children and employees often ban sites containing pornography, violence, and hate speech. What counts as pornography and violence? And what, exactly, is "hate speech?" This sounds like a phrase out of 1984.

しかし、今自分が生きている時代の流行を見極めるには、意識的な努力が必要だ。距離を置いてくれる時間というものがない以上、自分で距離を作らなければならない。群集の中に埋もれるのではなく、なるべくそこから離れたところに立って、皆が何をしているか眺めるんだ。そして、特に抑圧されているアイディアに注意を払う。子供や従業員向けの Web フィルタは通常、ポルノや暴力、煽動家の文章を遮断する。何がポルノや暴力と見なされているんだろう。そして、何がいったい「煽動」とされるんだろう。なんかまるで『1984 年』の世界から抜け出してきたような言葉じゃないか。

Labels like that are probably the biggest external clue. If a statement is false, that's the worst thing you can say about it. You don't need to say that it's heretical. And if it isn't false, it shouldn't be suppressed. So when you see statements being attacked as x-ist or y-ic (substitute your current values of x and y), whether in 1630 or 2030, that's a sure sign that something is wrong. When you hear such labels being used, ask why.

このようなレッテルは、最も有力で分かりやすい手がかりだ。もし主張が虚偽ならば、間違いだと指摘する以上に何かを言う必要はない。わざわざ主張を異端扱いする必要などないのだ。そして主張が虚偽でないのならば、抑圧されるべきではない。したがって、ある主張が「X-主義」だとか「Y-的」だとかいった言葉で攻撃されているならば(X と Y は適当な値で置換せよ)、それが 1630 年であろうと 2030 年であろうと、何かがおかしくなっていることの確かな兆候だ。そのようなレッテル貼りが行われているならば、なぜ、と問うてみるとよい。

Especially if you hear yourself using them. It's not just the mob you need to learn to watch from a distance. You need to be able to watch your own thoughts from a distance. That's not a radical idea, by the way; it's the main difference between children and adults. When a child gets angry because he's tired, he doesn't know what's happening. An adult can distance himself enough from the situation to say "never mind, I'm just tired." I don't see why one couldn't, by a similar process, learn to recognize and discount the effects of moral fashions.

特に自分がそういうレッテルを使ったときだ。距離を置いて見ることで学ぶべきものは、群集からだけではない。自分自身の考えも、距離を置いて注意深く見る必要がある。ところでこれは別に斬新なアイディアでもなんでもない。そもそも、それが子供と大人の主要

な違いだからだ。子供が疲れのせいで怒っていても、自分自身はどうして怒っているのかわからない。大人はそういう状況から一歩下がって自分を見て、「気にするな。疲れているだけなんだ」と言うことができる。全く同じようにして、道徳の流行の影響を認識して回避することを人が学べないということはあるまい。

You have to take that extra step if you want to think clearly. But it's harder, because now you're working against social customs instead of with them. Everyone encourages you to grow up to the point where you can discount your own bad moods. Few encourage you to continue to the point where you can discount society's bad moods.

明晰な思考をしたいのなら、その一歩を踏み出さなければならない。それは簡単ではない。それは社会的な慣習に沿うプロセスではなく、むしろ反するプロセスだからだ。自分自身の機嫌が悪いことに影響されないようにするところまでは、皆が教えてくれた。しかし、社会の機嫌が悪いことに影響されないようにするところまでやれと言ってくれる人はほとんどいない。

How can you see the wave, when you're the water? Always be questioning. That's the only defence. What can't you say? And why?

水の中にいる時に、どうやったら波を見ることができるだろう。常に問い続けることだ。それが唯一の防御だ。口にできないことは何だ?そしてなぜだ?

Notes

注釈

[1] The Inquisition probably never intended carry out their threat of torture. But that was because Galileo made it clear that he would do whatever they asked. If he had refused, it's hard to imagine they would simply have backed down. Not long before they had burnt the philosopher Giordano Bruno when he proved intransigent.

[1] ガリレイの異端審問では拷問の恐怖を与えるだけで、それを実行に移そうという意図は働いていなかったようだ。だが、それはガリレイが命令されたことは全て守るとはっきり言明したからだ。もしガリレイが拒否していたとしたら、教会がそのまま引き下がったとは思えない。そのしばらく前に、妥協を拒否した哲学者ジョルダノ・ブルーノは火あぶりにあっているのだ。

[2] Many organizations obligingly publish lists of what you can't say within them.

Unfortunately these are usually both incomplete, because there are things so shocking they don't even anticipate anyone saying them, and at the same time so general that they couldn't possibly be enforced literally. It's a rare university speech code that would not, taken literally, forbid Shakespeare.

[2] 多くの組織では、ご親切にも組織内で口にしてはならないことのリストを作成している。残念ながらそういうリストは、不完全だし、あまりに一般的過ぎる。ショッキング過ぎる言葉で、それを口にする人がいることすら想像できないものは、リストに載っていない。また、このようなリストはあまりに一般的過ぎて、文字通り強制することなど不可能という代物だ。大学の言葉のエチケット規則は、文字通り解釈したらシェークスピアも禁書になるようなものがほとんどだろう。

[3] Kundel HL, Nodine CF, Krupinski EA, "Searching for lung nodules: Visual dwell indicates locations of false-positive and false-negative decisions." *Investigative Radiology*, 24 (1989), 472-478.

[4] The verb "diff" is computer jargon, but it's the only word with exactly the sense I want. It comes from the Unix diff utility, which yields a list of all the differences between two files. More generally it means an unselective and microscopically thorough comparison between two versions of something.

[4] 「diffを取る」とはコンピュータ用語なんだが、ここで私が言いたいことを正確に表すのはこの言葉しかなかった。この言葉は Unix の diff プログラムから来ており、2つのファイルの差異をリストとして出してくれる。より一般的な意味としては、同種の2つのものをバイアス無しで詳細に比較するような場合に用いられる。

[5] It may seem from this that I am some kind of moral relativist. Far from it. I think that "judgemental" is one of the labels that gets used in our time to prevent discussion of ideas, and that our attempts to be "non-judgemental" will seem to future eras one of the most comical things about us.

[5] こう言うと、私が道徳に対して一種の相対主義者なんじゃないかと思われるかもしれない。実際はそれにはほど遠い。「審判的」というのは現代においてアイディアに関する議論を封じ込めるのに使われるようになったレッテルのひとつだ。私たちが「非審判的」であろうと努力する様子は、未来の世界から見て最高に滑稽に映るだろう。

[6] This makes the world confusing to kids, since what they see disagrees with what they're told. I could never understand *why*, for example, Portuguese "explorers" had started to work their way along the coast of Africa. In fact, they

were after slaves.

Bovill, Edward, *The Golden Trade of the Moors*, Oxford, 1963.

[6] これは子供にとって世界をかえって分かりにくいものにしてしまう。なぜなら教えられたものと実体が一致しないからだ。例えば、私はなんでポルトガル人たちの「探検」がアフリカの海岸沿いから開始されたのかがどうしても分からなかった。実は、彼らは奴隷を探していたのだ。

Bovill, Edward, *The Golden Trade of the Moors*, Oxford, 1963.

[7] The kids soon learn these words from their friends, but they know they're not supposed to use them. So for a while you have a state of affairs like something from a musical comedy, where the parents use these words among their peers, but never in front of the children, and the children use the words among their peers, but never in front of their parents.

[7] 子供はいずれこういう言葉を友人から得ていくが、それを使ってはいけない、ということも知っていく。したがって、ある期間の間、家族は一種のミュージカルコメディみたいな状況に置かれることになる。大人(親)は大人同士でそういう言葉を使うが、子供の前では決して使わない。一方、子供は子供同士でそういう言葉を使うが、親の前では決して使わないのだ。

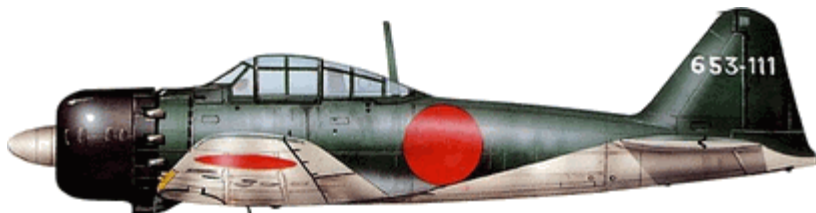
[8] A few years ago I worked for a startup whose [logo](#) was a solid red circle with a white V in the middle. I really liked this logo. After we'd been using it for a while, I remember thinking, you know, this is a really powerful symbol, a red circle. Red is arguably the most basic color, and the circle the most basic shape. Together they had such visual punch. Why didn't more American companies have a red circle as their logo? Ahh, [yes](#)...

[8] 数年前、私はベンチャー企業で働いていたが、そこのロゴは赤く塗りつぶした丸の真ん中に白い V があるものだった。私はこのロゴがとても気に入っていた。しばらくそれを使ってみて、こう思ったことを覚えている。これはとても強力なシンボルだよ、赤い丸っていうのは。赤はもちろん最も基本的な色の一つだし、丸は最も基本的な形だ。組み合わせられれば視覚的なインパクトは強烈だ。なんで米国企業はもっとロゴに赤い丸を使わないんだろう?あ、そうか...



VIAWEB FRONT PAGE

This is the Viaweb front page as it was in June 1998.



MITSUBISHI ZERO

[9] The fear is far the stronger of the two forces. Sometimes when I hear someone use the word "gyp" I tell them, with a serious expression, that one can't use that word anymore because it's considered disparaging to Romani (aka Gypsies). In fact dictionaries disagree about its etymology. But the reaction to this joke is nearly always one of slightly terrified compliance. There is something about fashion, in clothing or ideas, that takes away people's confidence: when they learn something new, they feel it was something they should have known already.

[9] この2つの力の間では、恐れの方がはるかに強い。誰かが「gyp(訳注：詐欺師など

を指す俗語)」という単語を使うのを聞くと、私は真面目な顔をして、それはロマニー(ジプシー)を貶める言葉だからもう使っちゃいけないんだよと言ってみることがある。辞書を見れば、語源的にはそんなことはないのだが、このジョークを聞いた人はほとんどいつも、真に受けてちょっとおびえたようなそぶりを見せる。流行には、それが服装に関するものであれ、思想に関するものであれ、人の自信を奪ってしまう何かがあるみたいだ。人は何か新しいことを知ると、もっと早くそのことを知っていればよかったと感じてしまうらしい。

[10] I don't mean to suggest that scientists' opinions are inevitably right, just that their willingness to consider unconventional ideas gives them a head start. In other respects they are sometimes at a disadvantage. Like other scholars, many scientists have never directly earned a living-- never, that is, been paid in return for services rendered. Most scholars live in an anomalous microworld in which money is something doled out by committees instead of a representation for work, and it seems natural to them that national economies should be run along the same lines. As a result, many otherwise intelligent people were socialists in the middle of the twentieth century.

[10] ここで、科学者の意見は常に正しいなんて言うつもりはない。ただ、常識にとらわれないアイデアを考えようとすることで、科学者は一歩先んじていると言いたいだけだ。別の面では、科学者は不利な点も背負っている。他の学者と同様、多くの科学者は生活の糧を直接稼ぐということは決してない。つまり、自分自身が行ったサービスの対価を直接受け取るということがない。多くの学者は、お金というものは仕事の対価ではなく委員会で分配されるものだという、風変わりな小宇宙の中に生きており、彼らにとって国家経済も同じように運営されていると考えるのは自然なのだ。結果として、他の点では聡明な人々の多くが、20世紀半ばには社会主義者であった。

[11] Presumably, within the industry, such thoughts would be considered "negative". Another label, much like "defeatist". Never mind that, one should ask, are they true or not? Indeed, the measure of a healthy organization is probably the degree to which negative thoughts are allowed. In places where great work is being done, the attitude always seems to be critical and sarcastic, not "positive" and "supportive". The people I know who do great work think that they suck, but that everyone else sucks even more.

[11] 業界の中では、おそらくそういう考えは「ネガティブ」なものだとされているんだろう。「敗北主義者」と同様のレッテルだ。そんなことは気にせずに、「それは真か偽か」と問うてみればいいんだ。実際、組織の健全さは、ネガティブな考えがどのくらい許容されるか

で測られるとも言えるんじゃないか。偉大な仕事になされる場所では、人々はいつも批評的で皮肉屋であり、決して「前向き」で「支援的」であったりしない。私が知っている、偉大な仕事をなした人は、自分はダメだが、他のみんなはもっとダメだ、と考えている。

[12] Behar, Richard, "The Thriving Cult of Greed and Power," *Time*, 6 May 1991.

[13] Healy, Patrick, "Summers hits 'anti-Semitic' actions," *Boston Globe*, 20 September 2002.

[14] "Tinkerers' champion," *The Economist*, 20 June 2002.

[15] By this I mean you'd have to become a professional controversialist, not that Noam Chomsky's opinions = what you can't say. If you actually said the things you can't say, you'd shock conservatives and liberals about equally-- just as, if you went back to Victorian England in a time machine, your ideas would shock Whigs and Tories about equally.

[15] ここで意味しているのは、言いたいことを全部言ったらあなたはフルタイムで議論をしている人にならざるを得ないということであって、ノーム・チョムスキーの意見が「口にできないこと」だと言っているのではない。チョムスキーの意見はリベラル派の主張にだいたい沿ったものだ。もしあなたが口にできないことを実際に口にしたら、保守派もリベラルも同様に衝撃を受けることだろう。あなたがビクトリア朝時代のイギリスにタイムマシンで旅行したとすれば、あなたの考えはホイッグ党とトーリー党の両方とも驚かしただろう。

[16] Traub, James, "Harvard Radical," *New York Times Magazine*, 24 August 2003.

[17] Miller, Arthur, *The Crucible in History and Other Essays*, Methuen, 2000.

Thanks to Sarah Harlin, Trevor Blackwell, Jessica Livingston, Robert Morris, Eric Raymond and Bob van der Zwaan for reading drafts of this essay, and to Lisa Randall, Jackie McDonough, Ryan Stanley and Joel Rainey for conversations about heresy. Needless to say they bear no blame for opinions expressed in it, and especially for opinions *not* expressed in it.

Chapter 4 THE WORD "HACKER"

第 4 章 天邪鬼の価値

April 2004



To the popular press, "hacker" means someone who breaks into computers. Among programmers it means a good programmer. But the two meanings are connected. To programmers, "hacker" connotes mastery in the most literal sense: someone who can make a computer do what he wants—whether the computer wants to or not.

一般報道では、「ハッカー」とはコンピュータに侵入する人物のことを指す。プログラマの間では、その言葉は優れたプログラマを指す。この 2 つの意味は実はつながっている。プログラマにとって「ハッカー」とは、文字通りその道の達人であることを意味しているんだ。つまり、コンピュータに、良いことであれ悪いことであれ、自分のやりたいことをやらせることができる者、ということだ。

To add to the confusion, the noun "hack" also has two senses. It can be either a compliment or an insult. It's called a hack when you do something in an ugly way. But when you do something so clever that you somehow beat the system, that's also called a hack. The word is used more often in the former than the latter sense, probably because ugly solutions are more common than brilliant ones.

混乱に拍車をかけているのが、名称の「ハック」にやはり 2 つの意味があるということだ。それは賞賛にもなるし、侮辱にもなる。何かをとて醜い方法でやったら、ハックと呼ばれる。しかし、何かを素晴らしく巧みな方法でやってのけてシステムをやっつけたなら、それもハックと呼ばれる。前者の意味で使われることのほうが後者の意味で使われることより多いが、それはたぶん、素晴らしい解答は醜い解答ほど多くはないということだろう。

Believe it or not, the two senses of "hack" are also connected. Ugly and imaginative solutions have something in common: they both break the rules. And there is a gradual continuum between rule breaking that's merely ugly (using duct

tape to attach something to your bike) and rule breaking that is brilliantly imaginative (discarding Euclidean space).

信じられないかもしれないけど、この「ハック」の 2 つの意味もつながっているんだ。醜い解答と独創的な解答には、共通点がある。両方とも、ルールを破っているということだ。そして実際、ルール破りの単なる醜いやり方(ガムテープでオートバイのどこかを修繕しておくとかね)と、ルール破りの素晴らしい独創性(ユークリッド空間を捨て去るとかね)とは、ひと続きのスペクトルの両端にある。

Hacking predates computers. When he was working on the Manhattan Project, Richard Feynman used to amuse himself by breaking into safes containing secret documents. This tradition continues today. When we were in grad school, a hacker friend of mine who spent too much time around MIT had his own lock picking kit. (He now runs a hedge fund, a not unrelated enterprise.)

ハッキングはコンピュータ以前からあった。リチャード・ファインマンはマンハッタンプロジェクトに参加していた時に、機密書類が保管されている金庫を破ることに楽しみを見出していた。こういう伝統は今日まで続いている。事実、私の大学院時代のハッカーの友人にも、ピッキング道具を一式揃えているやつがいた(彼は MIT 界限を長いことうろうろしていたが、今はヘッジファンドを経営している。なるほど、とうなずけるものがある)。

It is sometimes hard to explain to authorities why one would want to do such things. Another friend of mine once got in trouble with the government for breaking into computers. This had only recently been declared a crime, and the FBI found that their usual investigative technique didn't work. Police investigation apparently begins with a motive. The usual motives are few: drugs, money, sex, revenge. Intellectual curiosity was not one of the motives on the FBI's list. Indeed, the whole concept seemed foreign to them.

どうしてそういうことをしたがるのかを権力者に説明するのはとても難しい。私の別の友人は、コンピュータへの侵入のかどで政府ともめたことがある。それが犯罪とされるようになったのはつい最近のことだが、FBI は彼らの従来の捜査手法がうまく使えないことに気付いた。警察の捜査は通常、動機の考察から始まるようだ。普通、動機というのは限られている。麻薬とか、金とか、セックスとか、復讐とか。知的好奇心というのは FBI の動機のリストには載っていなかった。事実、知的好奇心そのものが彼らにとっては異質なものであったのだ。

Those in authority tend to be annoyed by hackers' general attitude of disobedience. But that disobedience is a byproduct of the qualities that make

them good programmers. They may laugh at the CEO when he talks in generic corporate new speech, but they also laugh at someone who tells them a certain problem can't be solved. Suppress one, and you suppress the other.

権力の側にある者は、こうしたハッカーの不服従の態度を警戒しがちだ。でも、不服従というのは、彼らを優れたプログラマにしている性質の副産物にすぎない。彼らは、例えば会社の定例会見での社長のスピーチを笑い飛ばすかもしれないが、誰かの「この問題は解けないよ」という言葉も笑い飛ばすだろう。片方を抑え込めば、もう一方も抑えられることになる。

This attitude is sometimes affected. Sometimes young programmers notice the eccentricities of eminent hackers and decide to adopt some of their own in order to seem smarter. The fake version is not merely annoying; the prickly attitude of these posers can actually slow the process of innovation.

ハッカーの態度は他人の影響を受けて生じることもある。若いプログラマが著名なハッカーの奇妙な振る舞いを見て、自分を賢く見せるためにそれを真似することはよくある。こういう形だけの態度は、ただ煩わしいというだけに留まらない。似非ハッカーたちの鼻につく態度は、変革のプロセスを遅らせることになりかねないんだ。

But even factoring in their annoying eccentricities, the disobedient attitude of hackers is a net win. I wish its advantages were better understood.

ハッカーには時に煩わしく感じられる奇妙な振る舞いがあるが、それを差し引いても、ハッカーの不服従の態度は大局的には得になる。その利点はもっと広く理解されてしかるべきだと思う。

For example, I suspect people in Hollywood are simply mystified by hackers' attitudes toward copyrights. They are a perennial topic of heated discussion on Slashdot. But why should people who program computers be so concerned about copyrights, of all things?

例えば、私は思うのだが、ハリウッドの連中はなんでハッカーが著作権の問題にうるさいのか不思議に思っているんじゃないだろうか。スラッシュドットではそのトピックが出るたびに熱い議論が巻き起こる。でも、一体全体どうしてまた、コンピュータに向かってプログラムを書いているような人々が著作権についてこんなに敏感なんだろうか。

Partly because some companies use mechanisms to prevent copying. Show any hacker a lock and his first thought is how to pick it. But there is a deeper reason that hackers are alarmed by measures like copyrights and patents. They see

increasingly aggressive measures to protect "intellectual property" as a threat to the intellectual freedom they need to do their job. And they are right.

理由のひとつは、いくつかの会社が著作権を、コピーを防ぐ機構として使っていることだろう。ハッカーに錠前を見せれば、きっと彼はそれを開けようとするはずだ。でも、ハッカーが著作権や特許といった考え方に警戒心を示すのは、もっと深い理由があるんだ。ハッカーたちは、強まりつつある「知的財産」保護の動きを、自分たちの仕事に必要な知的自由に対する脅威と考えている。そして、それは正しい。

It is by poking about inside current technology that hackers get ideas for the next generation. No thanks, intellectual homeowners may say, we don't need any outside help. But they're wrong. The next generation of computer technology has often—perhaps more often than not—been developed by outsiders. In 1977 there was no doubt some group within IBM developing what they expected to be the next generation of business computer. They were mistaken. The next generation of business computer was being developed on entirely different lines by two long-haired guys called Steve in a [garage](#) in Los Altos. At about the same time, the powers that be were cooperating to develop the official next generation operating system, Multics. But two guys who thought Multics excessively complex went off and wrote their own. They gave it a name that was a joking reference to Multics: Unix.

ハッカーは、現在の技術の中身を覗くことで次世代のアイディアを得る。知的財産の持ち主は、そんなの結構、外部の助けはいらないよ、と言うかもしれない。だが、それは間違っている。事実、コンピュータの次世代技術を見てみると、外部の人間により作られたもののほうが多いくらいなんだ。1977年には、IBM 内部から次世代のビジネスコンピュータが生まれると誰もが疑わなかった。だが、違った。次世代のビジネスコンピュータは、全く別の場所から現れた。ロスアルトスのガレージで、共にスティーブという名の 2 人の長髪の若者が創り出したんだ。それと同じころ、同様の力が、次世代のオペレーティングシステムとなるべき Multics に注がれていた。だが、Multics は複雑過ぎると考えた 2 人の若者が、自分たちの OS を書いて、Multics を茶化した名前を付けた。それが Unix だ。



APPLE GARAGE

This is the garage in Los Altos, CA, in which some of the first Apple computers were assembled. [Click for a larger version.](#)

The latest intellectual property laws impose unprecedented restrictions on the sort of poking around that leads to new ideas. In the past, a competitor might use patents to prevent you from selling a copy of something they made, but they couldn't prevent you from taking one apart to see how it worked. The latest laws make this a crime. How are we to develop new technology if we can't study current technology to figure out how to improve it?

最近の知的財産に関する法律は、新しいアイディアへとつながるような、あちこち覗きまわる行為に対して前例のないくらい厳しい制限を課す。かつては、他人に製品のコピーを作らせないように特許が使われたが、製品を分解してどう動いているかを調べることを禁じる方法はなかった。最近の法律はその行為さえ犯罪としてしまう。現在の技術を学び、それを改善してゆく方法を考えることができないとしたら、どうやって新しい技術を開発できるというのだろうか。

Ironically, hackers have brought this on themselves. Computers are responsible for the problem. The control systems inside machines used to be physical: gears and levers and cams. Increasingly, the brains (and thus the value) of products is in software. And by this I mean software in the general sense: i.e. data. A song on an LP is physically stamped into the plastic. A song on an iPod's disk is merely stored on it.

皮肉なことに、この問題を招いたのはハッカーたち自身だ。コンピュータがこの問題の原因だからだ。機械を制御する仕組みは、かつては歯車とかレバーとかカムのような物理的なものだった。次第に、製品の頭脳(とそれに伴って価値も)がソフトウェアへと移行してき

た。ここでのソフトウェアとは、広い意味での言葉だ。つまり、データということだ。LPレコードでは、歌はプラスチック上に物理的に刻印されていた。iPod では、歌はディスクに格納されているデータにすぎない。

Data is by definition easy to copy. And the Internet makes copies easy to distribute. So it is no wonder companies are afraid. But, as so often happens, fear has clouded their judgement. The government has responded with draconian laws to protect intellectual property. They probably mean well. But they may not realize that such laws will do more harm than good.

データは簡単にコピーできる。さらにインターネットはコピーをばらまくことを簡単にした。会社が恐れるのも無理はない。しかし、よくあることだが、恐怖が彼らの判断を曇らせてしまったんだ。政府はこの脅威に、知的財産を守る苛酷な法律で応えた。政府は良かれと思ってやっているんだろう。だが、そういう法律が善よりは害をなし得ることに彼らは気付いていない。

Why are programmers so violently opposed to these laws? If I were a legislator, I'd be interested in this mystery—for the same reason that, if I were a farmer and suddenly heard a lot of squawking coming from my hen house one night, I'd want to go out and investigate. Hackers are not stupid, and unanimity is very rare in this world. So if they're all squawking, perhaps there is something amiss.

なぜプログラマはこれらの法律に激しく反抗するのだろうか。私が法律の施行者だったとしたら、この奇妙な事実には興味を引かれるはずだ。ちょうど、農家で、ある晩突然鶏小屋で鶏たちが騒ぎだしたら、何が起こったか見に行くだろうというのと同じに。ハッカーは愚か者ではないし、ハッカーの世界で皆が同じ行動を取るというのは極めて珍しいことだ。だからハッカーが皆して騒いでいるということは、そこに何か不具合がある可能性が高い。

Could it be that such laws, though intended to protect America, will actually harm it? Think about it. There is something very American about Feynman breaking into safes during the Manhattan Project. It's hard to imagine the authorities having a sense of humor about such things over in Germany at that time. Maybe it's not a coincidence.

本来、米国を守るために作られたそういう法律が、実際には害をなしているということはある得るだろうか。考えてみよう。ファインマンがマンハッタンプロジェクトで金庫破りをしていたという事実は、とても米国的なところがある。例えば、当時、ドイツで権威ある人物がそのような真似をし得るだけのユーモアのセンスを持っていたとは考えにくい。これは偶然ではないかもしれない。

Hackers are unruly. That is the essence of hacking. And it is also the essence of American-ness. It is no accident that Silicon Valley is in America, and not France, or Germany, or England, or Japan. In those countries, people color inside the lines.

ハッカーは規則に従わない。それがハッキングの本質なんだ。それはまた、アメリカ的であることの本質でもある。シリコンバレーが、フランスやドイツやイギリスや日本ではなく米国に現れたのは偶然ではない。他の国々では、色を塗っても線からはみ出す人はいない。

I lived for a while in Florence. But after I'd been there a few months I realized that what I'd been unconsciously hoping to find there was back in the place I'd just left. The reason Florence is famous is that in 1450, it was New York. In 1450 it was filled with the kind of turbulent and ambitious people you find now in America. (So I went back to America.)

私はしばらくフィレンツェに住んだことがある。数ヵ月住んでみて、私が無意識のうちにそこで見つけようとしていたものは、私が後にしてきた場所にあったのだということに気付いた。フィレンツェが有名なのは、そこが 1450 年のニューヨークだったからだ。1450 年、そこは、今日米国に見られるような、不穏で野望を抱えた人々で溢れていた(だから私は米国に戻った)。

It is greatly to America's advantage that it is a congenial atmosphere for the right sort of unruliness—that it is a home not just for the smart, but for smartalecks. And hackers are invariably smart-alecks. If we had a national holiday, it would be April 1st. It says a great deal about our work that we use the same word for a brilliant or a horribly cheesy solution. When we cook one up we're not always 100% sure which kind it is. But as long as it has the right sort of wrongness, that's a promising sign. It's odd that people think of programming as precise and methodical. Computers are precise and methodical. Hacking is something you do with a gleeful laugh.

正当な理由のある不服従を許容する空気があることは、米国の大きな利点になっている。単に賢いだけでなく、ジョーク好きで小生意気な人々を惹きつけるからだ。ハッカーは例外なくジョーク好きで小生意気だ。ハッカー国の国家休日があるとすれば、それは四月一日になるだろう。素晴らしい解答にもおそろしくやつつけない仕事にも同じ言葉を使うという事実は、私たちのやっていることの本質をとてもうまく表現している。何かを作っているとき、それがどっちなのかは自分にも 100% 定かではないんだ。でも、それが正しい種

類のずれを持っていれば、何かすごいものになる可能性がある。奇妙なことに、プログラミングというものは正確で厳密なものだと思われているようだが、それは違う。正確で厳密なのはコンピュータのほうだ。ハッキング自体は、笑みを浮かべながら愉快にやるようなものだ。

In our world some of the most characteristic solutions are not far removed from practical jokes. IBM was no doubt rather surprised by the consequences of the licensing deal for DOS, just as the hypothetical "adversary" must be when Michael Rabin solves a problem by redefining it as one that's easier to solve.

コンピュータの世界では、最も賢い解というのは悪ふざけと紙一重であることがよくある。DOS のライセンスの結果起きたことに IBM は驚いたに違いない。ちょうど、マイケル・ラビンが問題を簡単なものに定義し直すことでパズルを解き、パズルの中の仮想敵を驚かせたように。

Smart-alecks have to develop a keen sense of how much they can [get away](#) with. And lately hackers have sensed a change in the atmosphere. Lately hackerliness seems rather frowned upon.

ジョーク好きで小生意気なハッカーたちは、どのくらいまで羽目を外してもよいかに関して極めて鋭いセンスを持っている。そして最近、彼らは空気が変わりつつあるのを感じている。ハッカー的なものに対して、眉をひそめられるようになってきたのだ。

To hackers the recent contraction in civil liberties seems especially ominous. That must also mystify outsiders. Why should we care especially about civil liberties? Why programmers, more than dentists or salesmen or landscapers?

ハッカーにとって、市民の自由権に対する最近の締め付けは、特に不気味なものに映る。このことも外部の人を不思議がらせるようだ。なんでハッカーが言論の自由に特に関心を持つんだろう。なぜ、歯医者やセールスマンや造園家よりも、プログラマがそういうことに関心を持つんだろう。

Let me put the case in terms a government official would appreciate. Civil liberties are not just an ornament, or a quaint American tradition. Civil liberties make countries rich. If you made a graph of GNP per capita vs. civil liberties, you'd notice a definite trend. Could civil liberties really be a cause, rather than just an effect? I think so. I think a society in which people can do and say what they want will also tend to be one in which the most efficient solutions win, rather than those sponsored by the most influential people. Authoritarian countries become

corrupt; corrupt countries become poor; and poor countries are weak. It seems to me there is a Laffer curve for government power, just as for tax revenues. At least, it seems likely enough that it would be stupid to try the experiment and find out. Unlike high tax rates, you can't repeal totalitarianism if it turns out to be a mistake.

政府の役人が喜ぶような言葉で状況を説明してみよう。自由権は飾りものでも、米国の古風な伝統でもない。自由権は国を富ませるものなんだ。国民一人当たり GNP と国民に許された自由とのグラフを書けば、はっきりとした傾向が見て取れるはずだ。自由権は結果じゃなくて、実は原因ではあるまいか。私はそう思う。自分の言いたいことが言え、やりたいことがやれる社会では、影響力のある人々に支持される解ではなく、最も効率の良い解が勝つ。権威主義が幅をきかせる国は腐敗する。腐敗した国は貧しくなる。そして貧しい国は弱くなる。税率と税収に関するラフター曲線と同じようなものが、政府の持つ力に対して見られると私は思う。少なくともその可能性は十分に高く、わざわざ試してどうなるか見てみるのは愚かなことだろう。高い税率とは違って、全体主義は間違っていると気付いても引き返すことができない。

This is why hackers worry. The government spying on people doesn't literally make programmers write worse code. It just leads eventually to a world in which bad ideas will win. And because this is so important to hackers, they're especially sensitive to it. They can sense totalitarianism approaching from a distance, as animals can sense an approaching thunderstorm.

これがハッカーが憂慮する理由だ。政府が国民を監視する国では、プログラマは文字通り悪いコードを書くんだ。それは、いずれ悪いアイディアが勝つ世界へとつながる。ハッカーにとってこのことはあまりに重要だから、特に敏感になるんだ。動物たちが嵐の気配を察知するように、ハッカーたちは全体主義が遠くから近づいて来るのを感じ取る。

It would be ironic if, as hackers fear, recent measures intended to protect national security and intellectual property turned out to be a missile aimed right at what makes America successful. But it would not be the first time that measures taken in an atmosphere of panic had the opposite of the intended effect.

ハッカーが恐れるように、国家の安全と知的財産を保護するために取られた最近の政策が、米国を成功に導いた要因へのミサイル攻撃であったことが分かったとしたら、皮肉なことだろう。だがパニックの空気が広がる時、意図した結果と反対の効果をもたらす政策が採られるのは、これが初めてではない。

There is such a thing as American-ness. There's nothing like living abroad to teach

you that. And if you want to know whether something will nurture or squash this quality, it would be hard to find a better focus group than hackers, because they come closest of any group I know to embodying it. Closer, probably, than the men running our government, who for all their talk of patriotism remind me more of Richelieu or Mazarin than Thomas Jefferson or George Washington.

米国的なもの、というのがある。外国に住むとそのことがよく分かる。その米国的な何かがこの性質を育てるか潰すかを知りたいければ、ハッカー以上のフォーカスグループはない。ハッカーたちは、私の知るどんな集団よりも、米国的なものを内包しているからだ。たぶん政府の閣僚たちよりも。権力者はいつも愛国心を語るが、それはトマス・ジェファークソンやジョージ・ワシントンよりも、リシュリューやマザランを思い起させる。

When you read what the founding fathers had to say for themselves, they sound more like hackers. "The spirit of resistance to government," Jefferson wrote, "is so valuable on certain occasions, that I wish it always to be kept alive."

私たちの国を築いた人々が、国を率いる人々について語った言葉は、むしろハッカーに似ている。ジェファークソンはこう書いている。「政府への反抗の精神は、ある種の状況では非常に価値のあるものだ。だから私は、そのような精神が常に保たれることを望む」

Imagine an American president saying that today. Like the remarks of an outspoken old grandmother, the sayings of the the founding fathers have embarrassed generations of their less confident successors. They remind us where we come from. They remind us that it is the people who break rules that are the source of America's wealth and power.

今日のアメリカ大統領がそういうことを言うと想像できるかい。年老いたおばあさんの忠告のように、国の創始者たちの言葉を聞いて、自信のない後継者たちは自らを恥じてきた。これらの言葉は、私たちがどこから来たのかを思い出させる。ルールを破る人々が、アメリカの富と力の源であるということ。

Those in a position to impose rules naturally want them to be obeyed. But be careful what you ask for. You might get it.

ルールを守らせる立場にある者は、他人がそれに従うことを自然に期待するだろう。だが、望むものに慎重であれ。望んだ通りのものが得られてしまうかもしれない。

Thanks to Ken Anderson, Trevor Blackwell, Daniel Giffin, Sarah Harlin, Shiro Kawai, Jessica Livingston, Matz, Jackie McDonough, Robert Morris, Eric Raymond, Guido van Rossum, David Weinberger, and Steven Wolfram for reading drafts of

this essay.

(The [image](#) shows Steve Jobs and Wozniak with a "blue box." Photo by Margret Wozniak. Reproduced by permission of Steve Wozniak.)

(鍵開け装置をいじっているジョブズとウォズニアック、1975 年)



JOBS AND WOZNIAC WITH A BLUE BOX

Before they made computers, the founders of Apple used to make hardware for hacking into the phone system. This photo was taken by Margret Wozniak in 1975.

コンピュータを作る前、アップルの創業者たちは電話機をハックするためのハードウェアを作っていた。1975 年、マルグレット・ウォズニアック撮影。

[Click to see a larger version.](#)

(Reproduced by permission of Steve Wozniak.)

Chapter 5 THE OTHER ROAD AHEAD

第 5 章 もうひとつの未来への道

September 2001

(This article explains why much of the next generation of software may be server-based, what that will mean for programmers, and why this new kind of software is a great opportunity for startups. It's derived from a talk at BBN Labs.)

(この記事では、次世代のソフトウェアの多くがサーバベースになるであろうこと、それがプログラマにとってどういう意味を持つかということ、そしてこの新しい種類のソフトウェアがベンチャーにとって重要なチャンスになる理由を説明する。この記事は、マサチューセッツ州ケンブリッジの BBN Labs で今年行った講演を元に作成したものである。)

In the summer of 1995, my friend Robert Morris and I decided to start a startup. The PR campaign leading up to Netscape's IPO was running full blast then, and there was a lot of talk in the press about online commerce. At the time there might have been thirty actual stores on the Web, all made by hand. If there were going to be a lot of online stores, there would need to be software for making them, so we decided to write some.

1995 年の夏、私は友人のロバート・モリスと共にベンチャー企業を立ち上げる決心をした。株式公開を目前にしたネットスケープの PR キャンペーンが大々的に展開され、「オンラインコマース」という言葉が報道でもてはやされていた頃のことだ。当時、Web 上で運営されていたオンラインストアの数は 30 やそこらだっただろうか、そのどれもが手作りだった。オンラインストアの数が大きく増えていくとしたら、サイトを構築するソフトウェアが必要とされるだろう、私たちはそう読んで開発に乗り出した。

For the first week or so we intended to make this an ordinary desktop application. Then one day we had the idea of making the software run on our Web server, using the browser as an interface. We tried rewriting the software to work over the Web, and it was clear that this was the way to go. If we wrote our software to run on the server, it would be a lot easier for the users and for us as well.

最初は普通のデスクトップアプリケーションを作るつもりだったが、1、2 週間ほど経ったある日、ひらめいた。Web サーバ上でソフトウェアを走らせたらどうだろう? Web ブラウザをインタフェースとして使えばいいのでは? Web で動くようにコードを書き直してみ、これが進むべき方向だとはっきりした。ユーザにとっても私たちにとっても全てが簡単になるからだ。

This turned out to be a good plan. Now, as [Yahoo Store](#), this software is the most popular online store builder, with about 14,000 users.

開発は成功裏に終わり、今、このソフトウェアは押しも押されもせぬ人気のオンラインストアビルダーとなっている。ユーザ数 1 万 4000 を超える Yahoo! Store がそれだ。

When we started Viaweb, hardly anyone understood what we meant when we said that the software ran on the server. It was not until Hotmail was launched a year later that people started to get it. Now everyone knows that this is a valid approach. There is a name now for what we were: an Application Service Provider, or ASP.

Viaweb を立ち上げた時、私たちが「サーバでソフトウェアを走らせるんだ」と説明しても、それがどういうことかを理解してくれる人はほとんどいなかった。1 年後、Hotmail がスタートし、その頃になってようやく人々は理解し始めた。今では、それが有効なアプローチだということは誰でも知っている。こういう業種に名前まで付いた。アプリケーションサービスプロバイダ、ASP というやつだ。

I think that a lot of the next generation of software will be written on this model. Even Microsoft, who have the most to lose, seem to see the inevitability of moving some things off the desktop. If software moves off the desktop and onto servers, it will mean a very different world for developers. This article describes the surprising things we saw, as some of the first visitors to this new world. To the extent software does move onto servers, what I'm describing here is the future. 次世代のソフトウェアの多くは、このモデルで書かれるようになるだろう。失うものが一番大きいはずのマイクロソフトでさえ、デスクトップからサーバへとシフトすることは避けられないと見ているようだ。ソフトウェアがデスクトップからサーバへシフトすれば、開発者の世界はずいぶん違ったものになるだろう。このエッセイでは、私たちがこの新世界の最初期の旅行者として得たびっくりするような経験を述べることにしたい。ソフトウェアをサーバへ移動させたという点においては、ここで述べていることは未来そのものである。

The Next Thing?

来たるべきもの？

When we look back on the desktop software era, I think we'll marvel at the inconveniences people put up with, just as we marvel now at what early car

owners put up with. For the first twenty or thirty years, you had to be a car expert to own a car. But cars were such a big win that lots of people who weren't car experts wanted to have them as well.

将来、デスクトップソフトウェアの時代を振り返った時、私たちはそれを使っていた人々に感心することだろう。ちょうど、昔の自動車なだめすかして乗り回していた人々に今、感心するようにね。最初の 20~30 年くらいは、車を持つためには車のエキスパートでなければならなかった。でも車はとても便利だったから、エキスパートでない人々も車を欲しがったんだ。

Computers are in this phase now. When you own a desktop computer, you end up learning a lot more than you wanted to know about what's happening inside it. But more than half the households in the US own one. My mother has a computer that she uses for email and for keeping accounts. About a year ago she was alarmed to receive a letter from Apple, offering her a discount on a new version of the operating system. There's something wrong when a sixty-five year old woman who wants to use a computer for email and accounts has to think about installing new operating systems. Ordinary users shouldn't even know the words "operating system," much less "device driver" or "patch."

コンピュータも今、ちょうどその時期にある。デスクトップコンピュータの持ち主になれば、知りたくもないその中身の動作をいやが応にも学ぶ羽目になる。でも、米国では半分以上の家庭にコンピュータがあるんだ。私の母だってコンピュータを使ってメールをやり取りし、家計簿を付けている。数年前、母はアップルから OS の新しいバージョンの割引購入特典を知らせるメールを受け取って困惑していた。メールと家計簿を使いたいだけの 65 歳の女性が、新しいオペレーティングシステムのインストールで悩まなくちゃならないなんて、どこかがおかしい。普通のユーザは、「オペレーティングシステム」なんて単語、知らなくても済むようじゃなくちゃダメだ。ましてや「デバイスドライバ」や「パッチ」だなんて。

There is now another way to deliver software that will save users from becoming system administrators. Web-based applications are programs that run on Web servers and use Web pages as the user interface. For the average user this new kind of software will be easier, cheaper, more mobile, more reliable, and often more powerful than desktop software.

今は、ユーザをシステム管理者にしなくてもよいソフトウェアの配布方法がある。Web ベースアプリケーションは、Web サーバ上で走り、Web ページをユーザインタフェースとするプログラムだ。平均的なユーザにとって、この新種のソフトウェアはデスクトップソフ

トウェアより簡単で安く、どこでも使えて、信頼性も高く、しかもパワフルなことさえある。

With Web-based software, most users won't have to think about anything except the applications they use. All the messy, changing stuff will be sitting on a server somewhere, maintained by the kind of people who are good at that kind of thing. And so you won't ordinarily need a computer, per se, to use software. All you'll need will be something with a keyboard, a screen, and a Web browser. Maybe it will have wireless Internet access. Maybe it will also be your cell phone. Whatever it is, it will be consumer electronics: something that costs about \$200, and that people choose mostly based on how the case looks. You'll pay more for Internet services than you do for the hardware, just as you do now with telephones. [\[1\]](#)

Web ベースソフトウェアでは、普通のユーザは自分が使うアプリケーション以外を気にしなくてよい。煩わしい事柄は全てどこかのサーバにあって、そういうことが得意な人が管理してくれている。だからソフトウェアを使うのに、いわゆるコンピュータというものを使う必要はないわけだ。キーボードと画面と Web ブラウザがついた何かがあればいい。無線のインターネットアクセスが付いているかもしれないし、携帯電話だっていいわけだ。何であれ、それはせいぜい 200 ドルくらいの、見た目が決め手で選ばれる家電製品になるだろう。やがてユーザは、ハードウェアよりインターネット上のサービスにお金を払うようになるだろう。今、電話がそうであるように。 [\[1\]](#)

It will take about a tenth of a second for a click to get to the server and back, so users of heavily interactive software, like Photoshop, will still want to have the computations happening on the desktop. But if you look at the kind of things most people use computers for, a tenth of a second latency would not be a problem. My mother doesn't really need a desktop computer, and there are a lot of people like her.

Web ベースでは、クリックしてサーバから返事が帰って来るのに 1/10 秒ほどかかるから、Photoshop のようなインタラクティブ性が重要なソフトウェアのユーザは、計算がデスクトップで行われることを望むだろう。でも、一般的なコンピュータの使い方を見れば、1/10 秒の遅延は問題にはならない。私の母はデスクトップコンピュータなんて必要としないし、母のような人々は他にもたくさんいる。

The Win for Users

ユーザにとってのメリット

Near my house there is a car with a bumper sticker that reads "death before inconvenience." Most people, most of the time, will take whatever choice requires least work. If Web-based software wins, it will be because it's more convenient. And it looks as if it will be, for users and developers both.

うちの近所に、こんなステッカーを貼った車が停めてある、「不便なら死んだほうがまし」。多くの人はほとんどの場合、一番手間のかからない方法を選ぶ。Web ベースのソフトウェアが勝つとすれば、それが便利だからに他ならないはずだ。そしてどうもそれは正しいようだ。ユーザにとっても開発者にとっても。

To use a purely Web-based application, all you need is a browser connected to the Internet. So you can use a Web-based application anywhere. When you install software on your desktop computer, you can only use it on that computer. Worse still, your files are trapped on that computer. The inconvenience of this model becomes more and more evident as people get used to networks.

純粋な Web ベースのアプリケーションを使うのに必要なのは、インターネットに接続されたブラウザだけだ。したがって、Web ベースアプリケーションはどこでも使うことができる。デスクトップコンピュータにソフトウェアをインストールした場合は、そのコンピュータでしかソフトウェアを使うことができない。さらに悪いことに、使うファイルはそのコンピュータに閉じ込められてしまう。人々がネットワークを使うようになるにつれ、このモデルの不便さはよりはっきりしてくる。

The thin end of the wedge here was Web-based email. Millions of people now realize that you should have access to email messages no matter where you are. And if you can see your email, why not your calendar? If you can discuss a document with your colleagues, why can't you edit it? Why should any of your data be trapped on some computer sitting on a faraway desk?

この^{くさび}の先端は、Web ベースのメールだ。何百万という人々がどこでもメールを読み書きできる必要性に気付いている。メールが読めるなら、予定表だって使えたらいいじゃないか。同僚と文書に関して議論できるなら、それを編集できたっていいじゃないか。どうしてデータをはるか遠くの机に鎮座しているコンピュータに閉じ込めておかなくちゃならないんだい？

The whole idea of "your computer" is going away, and being replaced with "your data." You should be able to get at your data from any computer. Or rather, any client, and a client doesn't have to be a computer.

「私のコンピュータ」という概念はそっくり、「私のデータ」に取って代わられるだろう。

どのコンピュータからも自分のデータにアクセスできなくちゃ。いや、むしろどのクライアントからでも、と言っておこうか。クライアントがコンピュータである必要はないからだ。

Clients shouldn't store data; they should be like telephones. In fact they may become telephones, or vice versa. And as clients get smaller, you have another reason not to keep your data on them: something you carry around with you can be lost or stolen. Leaving your PDA in a taxi is like a disk crash, except that your data is handed to [someone else](#) instead of being vaporized.

クライアントはデータを保存すべきではない。電話機のようになるべきだ。実際、クライアントは電話機になるかもしれない(逆かもしれないが)。そしてクライアントが小さくなってゆけば、ますますクライアントにデータを置いておきたくなくなるだろう。持ち運ぶものには紛失や盗難という危険性がある。PDA をタクシーに置き忘れるのは、ディスククラッシュと同じような事故だ。ただ、データが消えてしまうのではなく他人の手に渡るってところが違う。

With purely Web-based software, neither your data nor the applications are kept on the client. So you don't have to install anything to use it. And when there's no installation, you don't have to worry about installation going wrong. There can't be incompatibilities between the application and your operating system, because the software doesn't run on your operating system.

純粋な Web ベースのソフトウェアでは、データもアプリケーションもクライアント上には置かれない。だからそれを使うのにインストールする必要もない。インストールの必要がなければ、インストールがうまくいかなかったときの心配をする必要もない。OS とアプリケーションの非互換性の問題も起こらない。ソフトウェアは手元の OS 上で走るわけじゃないからね。

Because it needs no installation, it will be easy, and common, to try Web-based software before you "buy" it. You should expect to be able to test-drive any Web-based application for free, just by going to the site where it's offered. At Viaweb our whole site was like a big arrow pointing users to the test drive.

インストールが不要ということから、購入する前に Web ベースのソフトウェアを試してみることは簡単だし、おそらくそれが当たり前になってゆくだろう。どんな Web ベースアプリケーションも、そのサイトに行くだけで、無料で試用できるようになるだろう。Viaweb では、私たちのサイト全てがユーザを試用に誘導するようなでっかい矢印みたいなものだった。

After trying the demo, signing up for the service should require nothing more than filling out a brief form (the briefer the better). And that should be the last work the user has to do. With Web-based software, you should get new releases without paying extra, or doing any work, or possibly even knowing about it.

デモを試した後にサービスにサインアップするには、短いフォームに記入するだけでいい(短ければ短いほどいい)。そして、それがユーザにとって最後の作業であるべきだ。Webベースソフトウェアでは新しいリリースは追加料金も余分な作業もなしで使えるようになる。新しいリリースがあったことにさえ気付かないかもしれない。

Upgrades won't be the big shocks they are now. Over time applications will quietly grow more powerful. This will take some effort on the part of the developers. They will have to design software so that it can be updated without confusing the users. That's a new problem, but there are ways to solve it.

アップグレードは現在のように大がかりなものではなくなるだろう。時が経つにつれ、アプリケーションは静かに、より強力なものに成長してゆく。この点に関しては、開発者側にある程度の努力が必要だ。ソフトウェアの更新がユーザを混乱させないようにあらかじめ設計しておかねばならない。これは新しい問題ではあるが、解決法はある。

With Web-based applications, everyone uses the same version, and bugs can be fixed as soon as they're discovered. So Web-based software should have far fewer bugs than desktop software. At Viaweb, I doubt we ever had ten known bugs at any one time. That's orders of magnitude better than desktop software.

Webベースアプリケーションでは、誰もが同じバージョンを使い、バグは見つかるはしから修正されてゆく。したがって、Webベースアプリケーションはデスクトップソフトウェアよりもバグが少ないだろう。Viawebでは、たぶんどどの時点でも、既知のバグの数は10以下だったんじゃないかと思う。これはデスクトップソフトウェアとは桁違いの少なさだ。

Web-based applications can be used by several people at the same time. This is an obvious win for collaborative applications, but I bet users will start to want this in most applications once they realize it's possible. It will often be useful to let two people edit the same document, for example. Viaweb let multiple users edit a site simultaneously, more because that was the right way to write the software than because we expected users to want to, but it turned out that many did.

Webベースアプリケーションは複数のユーザから同時に使える。これは共同作業を行うアプリケーションにとって明確な利点だが、ユーザは、それが可能だと知ったら、ほとんど

のアプリケーションがそのように使えることを望むのではないだろうか。例えば、2人で同時に同じ文書を編集するのが便利なときは良くある。**Viaweb** では複数のユーザが同じサイトを同時に編集できた。ユーザが欲したからというより、そうするのがソフトウェアとして正しい方法だったからなのだが、結果的に多くのユーザがその機能を使いたがったのだ。

When you use a Web-based application, your data will be safer. Disk crashes won't be a thing of the past, but users won't hear about them anymore. They'll happen within server farms. And companies offering Web-based applications will actually do backups-- not only because they'll have real system administrators worrying about such things, but because an ASP that does lose people's data will be in big, big trouble. When people lose their own data in a disk crash, they can't get that mad, because they only have themselves to be mad at. When a company loses their data for them, they'll get a lot madder.

Web ベースアプリケーションを使えばデータはより安全だ。ディスククラッシュが過去のものになることはないだろうが、ユーザがそれを気にする必要はなくなる。クラッシュはサーバファーム内で起こるからだ。そして、Web ベースアプリケーションを提供する会社はちゃんとバックアップを取っているはずだ。ちゃんとしたシステム管理者がいてそういうことを気にかけているから、というだけではない。**ASP** にとってユーザのデータを失うということは、あまりに重大な問題だからだ。ユーザは、自分のディスクがクラッシュして自分のデータが消えたときにはそれほど怒らない。自分を責めるしかないからね。だが、**ASP** が自分のデータを消したとなればかんかんになるだろう。

Finally, Web-based software should be less vulnerable to viruses. If the client doesn't run anything except a browser, there's less chance of running viruses, and no data locally to damage. And a program that attacked the servers themselves should find them very well defended. [\[2\]](#)

最後に、Web ベースのソフトウェアはウイルスに対する耐性がより強いだろう。クライアントがブラウザだけを走らせるなら、ウイルスが走る危険はより少なくなるし、被害を受けるローカルデータは存在しない。一方、サーバを攻撃しようにも、ずっと守りが堅いはずだ。 [\[2\]](#)

For users, Web-based software will be *less stressful*. I think if you looked inside the average Windows user you'd find a huge and pretty much untapped desire for software meeting that description. Unleashed, it could be a powerful force.

ユーザにとって、Web ベースのソフトウェアはストレスの少ないものになるだろう。平均的な Windows ユーザの心の中には、ストレスの少ないソフトウェアに対する満たされない

欲求というものがいっぱい溜っているに違いないと私は思っている。解き放たればそれは巨大な力となるだろう。

City of Code

コードの都市

To developers, the most conspicuous difference between Web-based and desktop software is that a Web-based application is not a single piece of code. It will be a collection of programs of different types rather than a single big binary. And so designing Web-based software is like designing a city rather than a building: as well as buildings you need roads, street signs, utilities, police and fire departments, and plans for both growth and various kinds of disasters.

開発者にとって Web ベースとデスクトップソフトウェアとの最も顕著な違いは、Web ベースアプリケーションはひとつのコードの塊ではないということだ。それは単一の大きなバイナリではなく、むしろ違ったタイプのプログラムの集合になるだろう。したがって Web ベースアプリケーションの設計は、ビルディングを作るよりも都市を作るのに似てくる。ビルディングだけでなく、道路や標識、上下水道にガス電気、警察署と消防署、そして拡張計画や災害への対応計画などが必要だ。

At Viaweb, software included fairly big applications that users talked to directly, programs that those programs used, programs that ran constantly in the background looking for problems, programs that tried to restart things if they broke, programs that ran occasionally to compile statistics or build indexes for searches, programs we ran explicitly to garbage-collect resources or to move or restore data, programs that pretended to be users (to measure performance or expose bugs), programs for diagnosing network troubles, programs for doing backups, interfaces to outside services, software that drove an impressive collection of dials displaying real-time server statistics (a hit with visitors, but indispensable for us too), modifications (including bug fixes) to open-source software, and a great many configuration files and settings. Trevor Blackwell wrote a spectacular program for moving stores to new servers across the country, without shutting them down, after we were bought by Yahoo. Programs paged us, sent faxes and email to users, conducted transactions with credit card processors, and talked to one another through sockets, pipes, http requests, ssh, udp packets, shared memory, and files. Some of Viaweb even consisted of the absence of programs, since one of the keys to Unix security is not to run unnecessary utilities

that people might use to break into your servers.

Viaweb で使っていたソフトウェアには、ユーザが直接対話するかなり大きなアプリケーション群と、それらが使うプログラム群、そして問題を監視するためにバックグラウンドで常に走っているプログラム群、おかしくなった時に再立ち上げを行うプログラム群、時々走って統計情報をまとめたりサーチインデックスを構築するプログラム群、データの移動や資源のごみ集めのために手動で走らせるプログラム群、性能評価やバグ出しのためにユーザのエミュレーションを行うプログラム群、ネットワークのトラブルを診断するプログラム群、バックアップ、外部サービスへのインタフェース、リアルタイムでサーバの統計情報をなかなか見事なダイアルの集合で表示するプログラム(訪問者に好評だったが、私たちにとっても無くてはならないものだった)、オープンソースソフトウェアへの変更やバグフィクス、その他数え切れないくらいの設定ファイルなどが含まれていた。トレバー・ブラックウェルは、私たちが **Yahoo** に買収された後で、国の反対側にあるサーバへサービスを止めずにオンラインショッピングサイトを移動する見事なプログラムを書いた。プログラム群が私たちを呼び出し、ユーザにファックスやメールを送り、クレジットカード会社と取り引きを行った。プログラム群は互いにソケット、パイプ、**http** リクエスト、**ssh**、**udp** パケット、共有メモリ、ファイルを通じて会話をした。Viaweb の一部は、「プログラムが存在しないこと」により作られていた。というのも、外部からの侵入を試みる者が使える不必要なユーティリティを走らせないことが **Unix** のセキュリティでは重要だからだ。

It did not end with software. We spent a lot of time thinking about server configurations. We built the servers ourselves, from components-- partly to save money, and partly to get exactly what we wanted. We had to think about whether our upstream ISP had fast enough connections to all the backbones. We serially [dated](#) RAID suppliers.

これはソフトウェアだけにとどまらなかった。私たちはサーバ構成についても時間をかけて考えた。サーバを部品から自分たちで組み立てたんだ。費用を節約するためでもあり、また必要なものを的確に得るためでもあった。上流の **ISP** がバックボーンに十分速い接続を持っているかを考えなければならなかったし、いろいろなメーカーの **RAID** をとっかえひっかえしていた。

But hardware is not just something to worry about. When you control it you can do more for users. With a desktop application, you can specify certain minimum hardware, but you can't add more. If you administer the servers, you can in one step enable all your users to page people, or send faxes, or send commands by phone, or process credit cards, etc, just by installing the relevant hardware. We always looked for new ways to add features with hardware, not just because it

pleased users, but also as a way to distinguish ourselves from competitors who (either because they sold desktop software, or resold Web-based applications through ISPs) didn't have direct control over the hardware.

でも、ハードウェアは心配の種ってだけじゃない。ハードウェアを制御することができれば、ユーザに対してもっと多くを提供できるんだ。デスクトップアプリケーションでは、最低動作条件を指定することはできるけど、より高性能のハードウェアは当てにはできない。サーバを管理しているならば、必要なハードウェアをインストールするだけで、メッセージング機能とか、ファックス送信機能、電話で指示を出せる機能、あるいはクレジットカードの処理機能なんかを追加することができる。私たちはいつも、ハードウェアを追加することで新しい機能が追加できないかと考えていた。それはユーザを喜ばせるってだけじゃない。ライバル会社は、デスクトップソフトウェアを売ったり、Web ベースアプリケーションを **ISP** 経由で売ったりしていた。ハードウェアを直接制御できない彼らに対して、私たちのサービスを差別化する良い方法だったんだ。

Because the software in a Web-based application will be a collection of programs rather than a single binary, it can be written in any number of different languages. When you're writing desktop software, you're practically forced to write the application in the same language as the underlying operating system-- meaning C and C++. And so these languages (especially among nontechnical people like managers and VCs) got to be considered as the languages for "serious" software development. But that was just an artifact of the way desktop software had to be delivered. For server-based software you can use any language you want. [3] Today a lot of the top hackers are using languages far removed from C and C++: Perl, Python, and even Lisp.

Web ベースアプリケーションはひとつのバイナリではなくプログラムの集合であるということは、それをいくつもの異なる言語で書くことができる、ということでもある。デスクトップソフトウェアを書いている場合、現実的には、アプリケーションを下層のオペレーティングシステムと同じ言語で書かざるを得ない。つまり、C か C++ ってことだ。そのせいで、これらの言語は、特に管理職や投資家のような非技術者の間では、「本格的な」ソフトウェア開発の言語だと思われるようになったみたいだ。でもそれはデスクトップソフトウェアの開発の現状から生まれた副作用にすぎない。サーバベースのソフトウェアでは、望みのどんな言語でも使うことができる。[3] 今日では、優れたハッカーたちの多くが C や C++ からかけ離れた言語を使っている。それは、Perl、Python、あるいは Lisp だ。

With server-based software, no one can tell you what language to use, because you control the whole system, right down to the hardware. Different languages

are good for different tasks. You can use whichever is best for each. And when you have competitors, "you can" means "you must" (we'll return to this later), because if you don't take advantage of this possibility, your competitors will.

サーバベースのソフトウェアでは、誰の指示も受けず、自分の好きな言語を使うことができる。あなたがハードウェアに至るまで全てのシステムを制御しているからだ。異なる仕事に向いた、異なる言語がある。それぞれの仕事に一番向いた言語を使えばいい。もしライバルが現れたら、むしろあなたは積極的に異なる言語を使うべきだ(この点は後でまた述べよう)。だってあなたがそのチャンスを利用しなかったら、ライバルが利用してしまうだろうからね。

Most of our competitors used C and C++, and this made their software visibly inferior because (among other things), they had no way around the statelessness of CGI scripts. If you were going to change something, all the changes had to happen on one page, with an Update button at the bottom. As I've written elsewhere, by using [Lisp](#), which many people still consider a research language, we could make the Viaweb editor behave more like desktop software.

ライバルの多くは C や C++ を使っていて、そのせいで彼らのソフトウェアは(他の多くの欠点に加えて)見かけがみすぼらしかったし、CGI スクリプトが状態を持たないという点を回避できていなかった。何かを変えたければ、ページ内で全てを編集して、ページ下の「更新」ボタンを押すしかなかった。他のところで書いたように、多くの人には今でも研究用言語と考えられている Lisp を用いることで、私たちは Viaweb のエディタを、デスクトップソフトウェアのように振る舞わせることができたんだ。

Releases

リリース

One of the most important changes in this new world is the way you do releases. In the desktop software business, doing a release is a huge trauma, in which the whole company sweats and strains to push out a single, giant piece of code. Obvious comparisons suggest themselves, both to the process and the resulting product.

この新世界における最も重要な変化のひとつはリリース方法だ。デスクトップソフトウェアビジネスでは、リリースは大きな頭痛の種だ。全社を挙げて頑張って、ひとつの巨大なコードの塊を仕上げる。それと比べてみれば、開発プロセスと結果の製品の両方で、変化を見て取ることができる。

With server-based software, you can make changes almost as you would in a program you were writing for yourself. You release software as a series of incremental changes instead of an occasional big explosion. A typical desktop software company might do one or two releases a year. At Viaweb we often did three to five releases a day.

サーバベースのソフトウェアでは、自分用に使っているプログラムとほとんど同じような感覚で変更を加えることができる。ソフトウェアのリリースは、たまにある爆発のようなものではなく、段階的な変更の連続になる。普通のデスクトップソフトウェア会社であれば、リリース回数は年に 1~2 回といったところだろうか。Viaweb では、ときには 1 日に 3~5 回のリリースをすることもしばしばあった。

When you switch to this new model, you realize how much software development is affected by the way it is released. Many of the nastiest problems you see in the desktop software business are due to catastrophic nature of releases.

この新しいモデルに切り替えてみれば、ソフトウェアの開発プロセスがリリース方法にどれだけ影響されているかが分かるだろう。デスクトップソフトウェアビジネスの危険な問題の多くは、このリリース方法がもともと破滅へ向かう性質を抱えているってことに起因している。

When you release only one new version a year, you tend to deal with bugs wholesale. Some time before the release date you assemble a new version in which half the code has been torn out and replaced, introducing countless bugs. Then a squad of QA people step in and start counting them, and the programmers work down the list, fixing them. They do not generally get to the end of the list, and indeed, no one is sure where the end is. It's like fishing rubble out of a pond. You never really know what's happening inside the software. At best you end up with a statistical sort of correctness.

年に 1 回だけ新しいバージョンをリリースするとしたら、バグをまとめて処理しなくちゃならなくなりがちだ。リリース日のしばらく前に、半分がたコードを入れ換えた新しいバージョンをかき集める。その中には数え切れないほどのバグが隠れている。そこで品質管理部隊が登場してバグをリストアップし、プログラマがそれを片っ端から直していく。だいたい、リストの最後まで全部直しきれることなどまずない。実際、どこでリストが終わるのかさえ分からないことが多い。池をさらってがらくたを引き上げるようなものだ。ソフトウェアの中で正確に何が起きているかを完全に把握することはできないだろう。せいぜい、確率的に正しいという状態に持っていければ御の字だ。

With server-based software, most of the change is small and incremental. That in itself is less likely to introduce bugs. It also means you know what to test most carefully when you're about to release software: the last thing you changed. You end up with a much firmer grip on the code. As a general rule, you do know what's happening inside it. You don't have the source code memorized, of course, but when you read the source you do it like a pilot scanning the instrument panel, not like a detective trying to unravel some mystery.

サーバベースのソフトウェアでは、ほとんどの変更は小さく、段階的だ。それだけで、バグが忍び込む可能性が低くなる。また、リリースする前に何を重点的にテストすればよいのかも分かる。最後に変更したところだ。結果として、コードをしっかりと把握できる。中で何が起きているかがはっきり分かるんだ。もちろんソースコードを丸暗記するわけじゃないが、パイロットが計器盤を監視するみたいな感じでソースを読むことができるだろう。探偵が謎を解き明かすように読むんじゃないくてね。

Desktop software breeds a certain fatalism about bugs. You know that you're shipping something loaded with bugs, and you've even set up mechanisms to compensate for it (e.g. patch releases). So why worry about a few more? Soon you're releasing whole features you know are broken. [Apple](#) did this earlier this year. They felt under pressure to release their new OS, whose release date had already slipped four times, but some of the software (support for CDs and DVDs) wasn't ready. The solution? They released the OS without the unfinished parts, and users will have to install them later.

デスクトップソフトウェアでは、バグに関して、それがある種仕方のないものだという観念があった。バグだらけの製品を出荷して、例えばパッチリリースみたいにそれを埋め合せる方法を提供する。その後、新しいバグが出たからどうだっていうんだ? どうせ、もう少ししたら、新しい機能満載の(でも動かない)リリースを出すんだから。アップルは数年前それをやった。いくつかの機能(CD と DVD のサポート)がまだ終わってないのに、もう 4 回もリリースを延ばした新しい OS をどうしても出さなくちゃならないというプレッシャーに耐えられなかった。それで、どうなったのかって? アップルは未実装の部分を除いた OS をリリースした。そのため、ユーザはその部分を後でインストールしなくちゃならなかったんだ。

With Web-based software, you never have to release software before it works, and you can release it as soon as it does work.

Web ベースのソフトウェアなら、動く前にソフトウェアをリリースしなくちゃならないなんてことはないし、動いたらすぐにリリースできる。

The industry veteran may be thinking, it's a fine-sounding idea to say that you never have to release software before it works, but what happens when you've promised to deliver a new version of your software by a certain date? With Web-based software, you wouldn't make such a promise, because there are no versions. Your software changes gradually and continuously. Some changes might be bigger than others, but the idea of versions just doesn't naturally fit onto Web-based software.

この業界の経験が長い人はこう考えるかもしれない。動くまでソフトウェアをリリースしないってのは素敵だけど、ある日付までに特定のバージョンを出すって約束してたとしたらどうなるんだろう、と。Web ベースソフトウェアでは、そんな約束はしないんだ。だってバージョンってものがないんだから。ソフトウェアは徐々に、連続的に変化してゆく。大きい変更も小さい変更もあるだろうけれど、バージョンという考え方は Web ベースソフトウェアには馴染まない。

If anyone remembers Viaweb this might sound odd, because we were always announcing new versions. This was done entirely for PR purposes. The trade press, we learned, thinks in version numbers. They will give you major coverage for a major release, meaning a new first digit on the version number, and generally a paragraph at most for a point release, meaning a new digit after the decimal point.

Viaweb を覚えている人ならこれを聞いてちょっと疑問に思うかもしれないね。私たちはいつだって新しいバージョンってやつをアナウンスしていたから。あれは純粋に宣伝のためにやっていたんだ。業界のニュースを報道する人たちは、バージョン番号で考えるってことを私たちは知った。メジャーリリース、つまりバージョン番号の最初の桁が変われば、詳しく報道してくれる。マイナーリリース、つまりピリオドの後のバージョン番号の変更のときは、一段落ほどの扱いになる。

Some of our competitors were offering desktop software and actually had version numbers. And for these releases, the mere fact of which seemed to us evidence of their backwardness, they would get all kinds of publicity. We didn't want to miss out, so we started giving version numbers to our software too. When we wanted some publicity, we'd make a list of all the features we'd added since the last "release," stick a new version number on the software, and issue a press release saying that the new version was available immediately. Amazingly, no one ever called us on it.

Viaweb のライバルのうちの何社かはデスクトップソフトウェアを提供していて、実際にバージョン番号を付けていた。そしてリリースのたびに、私たちから見れば彼らが遅れていることの証明でしかないような事実が、大々的に報道されていた。しかし、乗り遅れているようには思われなくなかったので、私たちも自分のソフトウェアにバージョン番号を付けることにしたんだ。宣伝が欲しくなったら、その前の「リリース」から追加した機能のリストを作って、ソフトウェアに新しいバージョン番号を振って、「新バージョン直ちに出荷！」とプレスリリースを流したんだ。面白いことに、誰もそれについて文句を付けることはなかった。

By the time we were bought, we had done this three times, so we were on Version 4. Version 4.1 if I remember correctly. After Viaweb became Yahoo Store, there was no longer such a desperate need for publicity, so although the software continued to evolve, the whole idea of version numbers was quietly dropped.

会社を買収されるまでにそれを 3 回やったから、バージョン 4 までいったことになる。私の記憶が正しければ確かバージョン 4.1 だった。Viaweb が Yahoo Store になってからはそこまでして広告を打つ必要はなくなったから、ソフトウェア自体はどんどん進化していたけれど、バージョン番号なんてものはひっそりと消えてしまった。

Bugs

バグ

The other major technical advantage of Web-based software is that you can reproduce most bugs. You have the users' data right there on your disk. If someone breaks your software, you don't have to try to guess what's going on, as you would with desktop software: you should be able to reproduce the error while they're on the phone with you. You might even know about it already, if you have code for noticing errors built into your application.

Web ベースのソフトウェアのもうひとつの技術的な利点は、多くのバグを再現できるということだ。自分の手元のディスクにユーザのデータがあるわけだから。誰かがソフトウェアを壊してしまったとき、デスクトップソフトウェアの場合のように何が起きているかを推理する必要はない。ユーザの電話に対応している間に同じエラーを再現できるんだ。それどころか、アプリケーションにエラーを通知するコードを仕込んでおけば、ユーザが電話をかけてきた時には既にエラーの所在がわかっている、そんなことだって現実になり得るだろう。

Web-based software gets used round the clock, so everything you do is

immediately put through the wringer. Bugs turn up quickly.

Web ベースソフトウェアは一日中使われているから、どんな変更も直ちに試練に晒される。バグは素早く現れる。

Software companies are sometimes accused of letting the users debug their software. And that is just what I'm advocating. For Web-based software it's actually a good plan, because the bugs are fewer and transient. When you release software gradually you get far fewer bugs to start with. And when you can reproduce errors and release changes instantly, you can find and fix most bugs as soon as they appear. We never had enough bugs at any one time to bother with a formal bug-tracking system.

ソフトウェア会社はユーザにデバッグをさせていると非難されることがよくある。私がここで言っていることも、ユーザにデバッグをさせるということに他ならない。でも、Web ベースソフトウェアではそれはむしろ良い方策なんだ。バグは少ないし、ごく一時的なものだからだ。ソフトウェアを段階的にリリースすれば、はるかに少ないバグから始めることになるし、エラーを再現できて変更を直ちにリリースできるということは、ほとんどのバグは見つかるや否や修正されることになる。Viaweb では、バグトラッキングシステムなんて大げさなものが必要なほどバグがたまることは一度もなかった。

You should test changes before you release them, of course, so no major bugs should get released. Those few that inevitably slip through will involve borderline cases and will only affect the few users that encounter them before someone calls in to complain. As long as you fix bugs right away, the net effect, for the average user, is far fewer bugs. I doubt the average Viaweb user ever saw a bug.

もちろんリリースの前に変更箇所をテストするのは当然で、大きなバグがリリースされるようなことはあってはならない。ただ、どうしてもテストをすり抜けてしまったごく少数のバグがあったとしても、それに影響を受けるのは、苦情の電話がくるまでにそのバグに出会ってしまった少数のユーザだけだ。バグを直ちに修正するのであれば、全体として、平均的なユーザにとってはバグがはるかに少ないことになる。平均的な Viaweb ユーザは、バグに出会ったことさえなかったんじゃないかと思う。

Fixing fresh bugs is easier than fixing old ones. It's usually fairly quick to find a bug in code you just wrote. When it turns up you often know what's wrong before you even look at the source, because you were already worrying about it subconsciously. Fixing a bug in something you wrote six months ago (the average case if you release once a year) is a lot more work. And since you don't

understand the code as well, you're more likely to fix it in an ugly way, or even introduce more bugs. [\[4\]](#)

新鮮なバグを直すのは古いものを直すよりも簡単だ。書いたばかりのコードからバグを見つけるのは、たいていは素早くできる。それどころか、バグを見た瞬間に、ソースを見なくてもだいたいどこが悪いか見当がつくことさえある。そういうところは、意識下でなんとなく気になっているものだからだ。年 1 回のリリースなら、平均して 6 ヶ月前に書いたコードのバグを見つけることになるが、それはずっと難しい。そして、コードの理解が行き届かないうちに修正すれば、その修正は不格好なものになりがちで、それが別のバグを招いてしまうことだってある。 [\[4\]](#)

When you catch bugs early, you also get fewer compound bugs. Compound bugs are two separate bugs that interact: you trip going downstairs, and when you reach for the handrail it comes off in your hand. In software this kind of bug is the hardest to find, and also tends to have the worst consequences. [\[5\]](#) The traditional "break everything and then filter out the bugs" approach inherently yields a lot of compound bugs. And software that's released in a series of small changes inherently tends not to. The floors are constantly being swept clean of any loose objects that might later get stuck in something.

バグを早めに見つければ、複合したバグに悩まされることも少なくなる。複合したバグとは、別々のバグが相互に影響を及ぼしているようなものだ。まるで、階段を降りるときにつまずいて、あわてて手すりを掴んだらそれが外れてしまった、みたいな羽目になる。ソフトウェアではそういう類のバグは最も見つけにくく、また最悪の結果をもたらすことも多い。 [\[5\]](#) 伝統的な、「とりあえず壊れるにまかせて後でバグを直してけばいい」という方法は、本質的にたくさんの複合バグを生み出す。一方、小さな変更だけで次々にリリースされるソフトウェアは、本質的に複合バグが出にくい。床は常にきれいに掃かれていて、後で邪魔になりそうなものはきちんと片付けられている。

It helps if you use a technique called functional programming. Functional programming means avoiding side-effects. It's something you're more likely to see in research papers than commercial software, but for Web-based applications it turns out to be really useful. It's hard to write entire programs as purely functional code, but you can write substantial chunks this way. It makes those parts of your software easier to test, because they have no state, and that is very convenient in a situation where you are constantly making and testing small modifications. I wrote much of Viaweb's editor in this style, and we made our scripting language, [RTML](#), a purely functional language.

関数プログラミングと呼ばれる手法を使うのも役に立つ。関数的プログラミングとは、副作用を避けることだ。商用ソフトウェアよりは研究論文でよく目にしそうな言葉だけど、Web ベースアプリケーションではこれが非常に役に立つことがわかった。全てのプログラムを純粋に関数的に書くのは難しいけれど、かなり多くの部分を関数的に書くことはできる。そういう部分は状態を持たないから、独立してテストするのが簡単だ。小さな変更を常にテストしているような環境ではこれはとても便利だ。私は **Viaweb** のエディタの大部分をこのスタイルで書いたし、私たちの作った専用スクリプティング言語である **RTML** は、純粋関数型言語だった。

People from the desktop software business will find this hard to credit, but at Viaweb bugs became almost a game. Since most released bugs involved borderline cases, the users who encountered them were likely to be advanced users, pushing the envelope. Advanced users are more forgiving about bugs, especially since you probably introduced them in the course of adding some feature they were asking for. In fact, because bugs were rare and you had to be doing sophisticated things to see them, advanced users were often proud to catch one. They would call support in a spirit more of triumph than anger, as if they had scored points off us.

デスクトップソフトウェアビジネスから来た人には信じ難いことだろうけれど、**Viaweb** ではバグはほとんど狩りの獲物みたいなものになっていった。リリースに紛れ込んでしまったバグのほとんどは境界的な場合だったため、バグを見つけるのは、限界を押し広げようとしている進んだユーザであることが多かった。進んだユーザはバグに対してより寛容だし、特にそのバグがユーザの望んだ機能を追加する過程で入ったものであればなおさらだ。実際、バグが減多になかったから、バグを見るためにはかなり高度なことをやらなくちゃならなかったんで、進んだユーザはバグを見つけたことを自慢したくらいだ。サポートにかかって来る電話は、怒りにまかせたものよりも、まるで私たちから一本取った、みたいな勝利の響きを持ったものが多かった。

Support

サポート

When you can reproduce errors, it changes your approach to customer support. At most software companies, support is offered as a way to make customers feel better. They're either calling you about a known bug, or they're just doing something wrong and you have to figure out what. In either case there's not much you can learn from them. And so you tend to view support calls as a pain in the ass

that you want to isolate from your developers as much as possible.

エラーを再現することができると、カスタマーサポートの方法も違ってくる。多くのソフトウェア会社では、サポートはどっちかっていうと顧客の機嫌をとりなすためのようなものだ。ユーザは、既知のバグについてかけてくるか、何か間違った操作をしてあなたにそれを解決してもらいたがってる。どっちにせよ、サポートに電話をしてきたユーザからあなたが学べることはあまりない。結果として、サポートの電話は苦痛の根源になり、開発者をそんなものからはなるべく離しておきたいと思うようになる。

This was not how things worked at Viaweb. At Viaweb, support was free, because we wanted to hear from customers. If someone had a problem, we wanted to know about it right away so that we could reproduce the error and release a fix. Viaweb では全く違っていた。Viaweb ではサポートは無料で、というのもユーザからのフィードバックは大歓迎だったからだ。誰かが問題に当たったら、私たちはすぐにそれを知りたかった。そうすればエラーを再現できて、修正をリリースできるからだ。

So at Viaweb the developers were always in close contact with support. The customer support people were about thirty feet away from the programmers, and knew that they could always interrupt anything with a report of a genuine bug. We would leave a board meeting to fix a serious bug.

だから Viaweb では開発者はいつもサポート部門と緊密に連携していた。カスタマーサポート部門はプログラマから 10m と離れていないところにいて、バグだとはっきり分かっていることがらに関してはいつでもプログラマの仕事に割り込むことができた。バグが重大なものなら、取締役会だって抜け出したものだ。

Our approach to support made everyone happier. The customers were delighted. Just imagine how it would feel to call a support line and be treated as someone bringing important news. The customer support people liked it because it meant they could help the users, instead of reading scripts to them. And the programmers liked it because they could reproduce bugs instead of just hearing vague second-hand reports about them.

私たちのサポートのやり方はみんなを幸せにした。ユーザは喜んだ。サポートに電話して、まるで重要なニュースをもたらしてくれた人のように扱われることを想像してみてほしい。カスタマーサポートスタッフはその仕事が気に入った。ただ与えられたマニュアルを読み上げるのではなく、実際に問題解決のプロセスにかかわることができたからだ。そしてプログラマもそれが気に入った。また聞きの意味不明なバグレポートなんかではなく、実際にバグを再現することができたからだ。

Our policy of fixing bugs on the fly changed the relationship between customer support people and hackers. At most software companies, support people are underpaid human shields, and hackers are little copies of God the Father, creators of the world. Whatever the procedure for reporting bugs, it is likely to be one-directional: support people who hear about bugs fill out some form that eventually gets passed on (possibly via QA) to programmers, who put it on their list of things to do. It was very different at Viaweb. Within a minute of hearing about a bug from a customer, the support people could be standing next to a programmer hearing him say "Shit, you're right, it's a bug." It delighted the support people to hear that "you're right" from the hackers. They used to bring us bugs with the same expectant air as a cat bringing you a mouse it has just killed. It also made them more careful in judging the seriousness of a bug, because now their honor was on the line.

バグをその場で修正するという私たちの方針は、カスタマーサポートスタッフとハッカーの関係を変えた。多くのソフトウェア会社では、サポート要員は安い給料で雇われた人間の盾で、一方ハッカーは偉大なる神の小さき分身、世界の創造者だ。バグレポートの方法が何であれ、それは一方通行のものになりがちだ。サポートスタッフはバグのことを聞いて一定の様式書類を埋め、いずれ(品質管理部門を通ることもあるが)それがプログラマの手元に届き、プログラマはそれをやるべきことのリストに加える。Viawebでは様子は大きく違っていた。ユーザからのバグ報告が上がって来て 1 分もたたないうちに、サポートスタッフはプログラマの横に立っていて、プログラマが「あちゃー、君の言う通りだ。これはバグだね」というセリフを聞いている。ハッカーから「君の言う通りだ」と聞くことを、サポートスタッフは大いに喜んだ。彼らの以前の経験では、バグをプログラマのところに報告しに行くと、まるで猫が殺したての鼠をくわえてきたかのような顔をされていたのだ。このことによって、サポート要員はバグの重大性についてより注意深くなった。自分の名誉がかかっているわけだから。

After we were bought by Yahoo, the customer support people were moved far away from the programmers. It was only then that we realized that they were effectively QA and to some extent marketing as well. In addition to catching bugs, they were the keepers of the knowledge of vaguer, buglike things, like features that confused users. [6] They were also a kind of proxy focus group; we could ask them which of two new features users wanted more, and they were always right. Yahoo に買収された後、カスタマーサポートはプログラマから離れてしまった。実はそうやって初めて、はカスタマーサポートとは実質的に品質管理部門であり、また営業部門で

もあるということに私たちは気付いたんだ。バグを捕まえるのに加えて、サポート要員は「バグに近いような曖昧なこと」、例えばユーザを混乱させるような機能についての知識を蓄えていた。[\[6\]](#) また、ある意味で彼らはフォーカスグループの代理でもあった。2つの機能のどちらがユーザに望まれているかを尋ねると、いつでも正しい答えが返ってきたからだ。

Morale

士気

Being able to release software immediately is a big motivator. Often as I was walking to work I would think of some change I wanted to make to the software, and do it that day. This worked for bigger features as well. Even if something was going to take two weeks to write (few projects took longer), I knew I could see the effect in the software as soon as it was done.

ソフトウェアを直ちにリリースできるということは大きな励みになった。出勤する途中でソフトウェアに加えたい変更を考えて、その日のうちに実装してしまうことがよくあった。大きな機能でもこれはうまくいった。書くのに2週間かかるようなものでさえ(それ以上かかるプロジェクトなんてほとんどなかった)、できあがったら直ちにその効果を見られると知っていたからだ。

If I'd had to wait a year for the next release, I would have shelved most of these ideas, for a while at least. The thing about ideas, though, is that they lead to more ideas. Have you ever noticed that when you sit down to write something, half the ideas that end up in it are ones you thought of while writing it? The same thing happens with software. Working to implement one idea gives you more ideas. So shelving an idea costs you not only that delay in implementing it, but also all the ideas that implementing it would have led to. In fact, shelving an idea probably even inhibits new ideas: as you start to think of some new feature, you catch sight of the shelf and think "but I already have a lot of new things I want to do for the next release."

効果を見るのに1年後のリリースを待たねばならなかったとしたら、これらのアイディアの多くを、少なくともしばらくの間はどこかにしまっていたことだろう。でも、アイディアが大事なのは、それがもっと多くのアイディアに結び付くからだ。腰を据えて何かを書き始めて、書き上がってみたらその中のアイディアの半分は書きながら思い付いたことだった、っていう経験はないかい?ソフトウェアにもそういうことがある。ひとつのアイディアを実装してゆく過程で、もっとたくさんのアイディアがひらめくんだ。だから、アイデ

アイデアをしまっておくことは、ただ単にその実装が遅れるだけでなく、それを実装することで得られたであろう全てのアイデアを失うことになる。実際、アイデアを寝かせておくことは、新しいアイデアを得る邪魔にさえなり得る。新しい機能を考え始めるたびに、やるべきことがたまっている棚が目に入って、「うーん、でも次のリリースまでにやりたいことがもうこんなにあるんだよな」となるからだ。

What big companies do instead of implementing features is plan them. At Viaweb we sometimes ran into trouble on this account. Investors and analysts would ask us what we had planned for the future. The truthful answer would have been, we didn't have any plans. We had general ideas about things we wanted to improve, but if we knew how we would have done it already. What were we going to do in the next six months? Whatever looked like the biggest win. I don't know if I ever dared give this answer, but that was the truth. Plans are just another word for ideas on the shelf. When we thought of good ideas, we implemented them.

大企業が、機能を実装する代わりにやることは、それを計画することだ。Viaweb でもこの点に関して時々問題になった。投資家やアナリストは、私たちの将来の計画についてよく質問した。でも正直なところ、我々には何も計画がなかった。そりゃあ改善したい点についての漠然としたアイデアはあったけれども、それをどうやるかがわかった時点では、もうその実装が終わっている状態だった。次の6ヶ月で何をするかって?何であれ最大のメリットが得られるようなものさ。もちろんそんなことを言い出す勇気は無かったけれど、それが正直なところだった。「計画」っていうのは、「棚に積まれたアイデア」を言い換えたものだ。良いアイデアを思い付いたら、私たちはすぐにそれを実装していた。

At Viaweb, as at many software companies, most code had one definite owner. But when you owned something you really owned it: no one except the owner of a piece of software had to approve (or even know about) a release. There was no protection against breakage except the fear of looking like an idiot to one's peers, and that was more than enough. I may have given the impression that we just blithely plowed forward writing code. We did go fast, but we thought very carefully before we released software onto those servers. And paying attention is more important to reliability than moving slowly. Because he pays close attention, a Navy pilot can land a 40,000 lb. aircraft at 140 miles per hour on a pitching carrier deck, at night, more safely than the average teenager can cut a bagel.

Viaweb では、他の多くのソフトウェア会社と同じように、コードの多くは一人の所有者が決まっていた。でも、何かを所有するってことは、本当に所有することを意味していた。そのソフトウェアの所有者は、他の誰にもリリースについて許可を得る必要はなかったし、

そのリリースがあることさえ知らせる必要がなかった。何かを壊してしまうことに対する安全ネットはただ一つ、そんなことをしたら同僚に対して恥ずかしいってことで、それで十分だったんだ。今までの私の話で、我々はやみくもにソフトウェアを書き続けていたような印象を持たれたかもしれない。私たちは確かに速く進んでいたけれど、サーバにソフトウェアをリリースする前には十分に注意深く考えたんだ。信頼性にとって重要なのは、ゆっくり動くことよりも、注意を払っておくことだ。十分に注意を払っているからこそ、空軍のパイロットは、普通のティーンエイジャーがベークルを切るのよりもっと安全に、時速 200 キロで飛んでいる 18 トンもある戦闘機で、傾いた空母のデッキに夜間着陸をやっているのけるんだ。

This way of writing software is a double-edged sword of course. It works a lot better for a small team of good, trusted programmers than it would for a big company of mediocre ones, where bad ideas are caught by committees instead of the people that had them.

こんなソフトウェアの書き方はもちろん両刃の剣だ。信頼できる、良いプログラマからなる小さなチームでのほうが、大企業での平凡なプログラマからなるチームよりもうまくいく。後者では、悪いアイディアは、それを考えた人が気付いて直すのではなく、委員会まで先送りされてしまうからだ。

Brooks in Reverse

Brooks の逆転

Fortunately, Web-based software does require fewer programmers. I once worked for a medium-sized desktop software company that had over 100 people working in engineering as a whole. Only 13 of these were in product development. All the rest were working on releases, ports, and so on. With Web-based software, all you need (at most) are the 13 people, because there are no releases, ports, and so on. 幸いなことに、Web ベースソフトウェアは少数のプログラマしか必要としない。私は中規模のデスクトップソフトウェア会社で働いたことがある。そこには技術部門全体で 100 人以上のスタッフがいた。そのうち 13 人だけが製品開発に携わっていた。残りの皆は、リリースとか移植とか、そういうことをやっていた。Web ベースソフトウェアでは、必要なのは最大でもこの 13 人だけってことになる。リリースやら移植やらの作業がいらないわけだから。

Viaweb was written by just three people. [\[7\]](#) I was always under pressure to hire more, because we wanted to get bought, and we knew that buyers would have a

hard time paying a high price for a company with only three programmers. (Solution: we hired more, but created new projects for them.)

Viaweb は 3 人で書いた。[\[7\]](#) 私はいつでも、もっと人を雇わなくちゃならないんじゃないかというプレッシャーを感じていた。というのも、いずれ買収されることを望んでいたんだが、プログラマがたった 3 人しかいない会社に高い金を払う買い手はあまりいないからだ。(解決策：私たちはもっとプログラマを雇ったけれど、その分、新しいプロジェクトを作ったんだ)。

When you can write software with fewer programmers, it saves you more than money. As Fred Brooks pointed out in *The Mythical Man-Month*, adding people to a project tends to slow it down. The number of possible connections between developers grows exponentially with the size of the group. The larger the group, the more time they'll spend in meetings negotiating how their software will work together, and the more bugs they'll get from unforeseen interactions. Fortunately, this process also works in reverse: as groups get smaller, software development gets exponentially more efficient. I can't remember the programmers at Viaweb ever having an actual meeting. We never had more to say at any one time than we could say as we were walking to lunch.

ソフトウェアをより少ない人数で書くことができれば、人件費以上のものが節約できる。フレッド・ブルックスが「人月の神話」で指摘したように、プロジェクトに人員を追加することは、プロジェクトをかえって遅れさせがちだ。開発者間の可能な連携の数はグループの大きさに対して指数的に増大する。グループが大きくなればなるほど、スタッフは各ソフトウェアがどう協調すべきかを話し合うミーティングにより多くの時間をとられ、意図しなかった相互作用から生まれるバグも増えて行く。幸いなことに、このプロセスは逆方向にも動くんだ。グループが小さくなればなるほど、ソフトウェアの開発効率は指数的に増大する。Viaweb で、プログラマがミーティングというものをしたことがあったか、思い出せない。昼食での会話以上に話し合わなければならないことを抱えていたことなんてなかった。

If there is a downside here, it is that all the programmers have to be to some degree system administrators as well. When you're hosting software, someone has to be watching the servers, and in practice the only people who can do this properly are the ones who wrote the software. At Viaweb our system had so many components and changed so frequently that there was no definite border between software and infrastructure. Arbitrarily declaring such a border would have constrained our design choices. And so although we were constantly hoping that

one day ("in a couple months") everything would be stable enough that we could hire someone whose job was just to worry about the servers, it never happened.

欠点があるとすれば、それは全てのプログラマがある程度はシステム管理者でもなければならないということだ。ソフトウェアをホストしていれば、誰かがサーバを監視していなければならない。そして、現実にはその仕事に適任な唯一のスタッフは、ソフトウェアを開発した本人に他ならない。**Viaweb** ではシステムは膨大なコンポーネントから構成されていたし、変更も非常に多かったので、ソフトウェアとインフラとの間に明確な境界線はなかった。そこに適当な境界線を設けることは、私たちのデザイン上の選択の足かせになっただろう。私たちも、いつの日か(「あと 2 ヶ月のうちには…」)全てが落ち着いて、サーバを監視するためだけのスタッフを雇いたいと望んでいたけれど、それはついにならなかった。

I don't think it could be any other way, as long as you're still actively developing the product. Web-based software is never going to be something you write, check in, and go home. It's a live thing, running on your servers right now. A bad bug might not just crash one user's process; it could crash them all. If a bug in your code corrupts some data on disk, you have to fix it. And so on. We found that you don't have to watch the servers every minute (after the first year or so), but you definitely want to keep an eye on things you've changed recently. You don't release code late at night and then go home.

製品を積極的に開発している限り、別の選択肢はなかったと思う。**Web** ベースソフトウェアは、一日の分を書いてチェックインして家に帰る、みたいなものにはならない。それは生きていて、今まさにサーバ上で走っているんだ。最悪のバグがあれば、それは一人のユーザのプロセスだけでなく、全てのシステムをクラッシュさせてしまうかもしれない。自分のコードのバグがディスク上の何かのデータを壊したら、自分で直さなくちゃならない。全てがそんな具合だ。1 年ほど運用したら、そう神経質にサーバを常時監視している必要はないことは分かったけれど、それでも最近更新したものに関しては注意を怠ることはできない。夜遅くにコードをリリースしてそのまま家に帰ってしまう、ということはいないんだ。

Watching Users

ユーザを観察する

With server-based software, you're in closer touch with your code. You can also be in closer touch with your users. Intuit is famous for introducing themselves to customers at retail stores and asking to follow them home. If you've ever watched

someone use your software for the first time, you know what surprises must have awaited them.

サーバベースのソフトウェアでは、コードにより密に接していることができる。それだけでなく、ユーザにもより密に接することができるんだ。Intuit は、小売店で製品を買う客に、マーケットリサーチが家まで同行してインストールするところを見せてもらえないかと頼むことで有名だった。自作のソフトウェアをユーザが最初に触る様子を見たことがあるなら、彼らが顧客の自宅でびっくりするようなことを目にしたであろうとは想像に難くない。

Software should do what users think it will. But you can't have any idea what users will be thinking, believe me, until you watch them. And server-based software gives you unprecedented information about their behavior. You're not limited to small, artificial focus groups. You can see every click made by every user. You have to consider carefully what you're going to look at, because you don't want to violate users' privacy, but even the most general statistical sampling can be very useful.

ソフトウェアはユーザが期待するように動作すべきだ。でもユーザが何を期待するかなんて、実際にユーザを観察してみるまでは見当もつかないんだ。サーバベースソフトウェアはユーザの振る舞いについて、他では得難い情報を与えてくれる。作為的な小さいフォーカスグループに限定されることはない。全てのユーザが行う、全てのクリックを見ることができるんだ。もちろんユーザのプライバシーを侵すことはできないから、何を見るかについてはじっくり考えなくちゃならないけど、ごく一般的な統計的サンプリングだけでもとても有用だ。

When you have the users on your server, you don't have to rely on benchmarks, for example. Benchmarks are simulated users. With server-based software, you can watch actual users. To decide what to optimize, just log into a server and see what's consuming all the CPU. And you know when to stop optimizing too: we eventually got the Viaweb editor to the point where it was memory-bound rather than CPU-bound, and since there was nothing we could do to decrease the size of users' data (well, nothing easy), we knew we might as well stop there.

例えば、ユーザが実際にサーバ上にいると、ベンチマークに頼る必要はなくなる。ベンチマークはシミュレートされたユーザだ。サーバベースソフトウェアでは、実際のユーザを観察することができる。何を最適化すればいいかを見つけるには、単にサーバにログインして、CPU を食いつぶしているのがいったい何かを見るだけでいい。どこで最適化を止めたらいいかを知ることでもある。私たちは Viaweb のエディタを、CPU ではなくメモリが

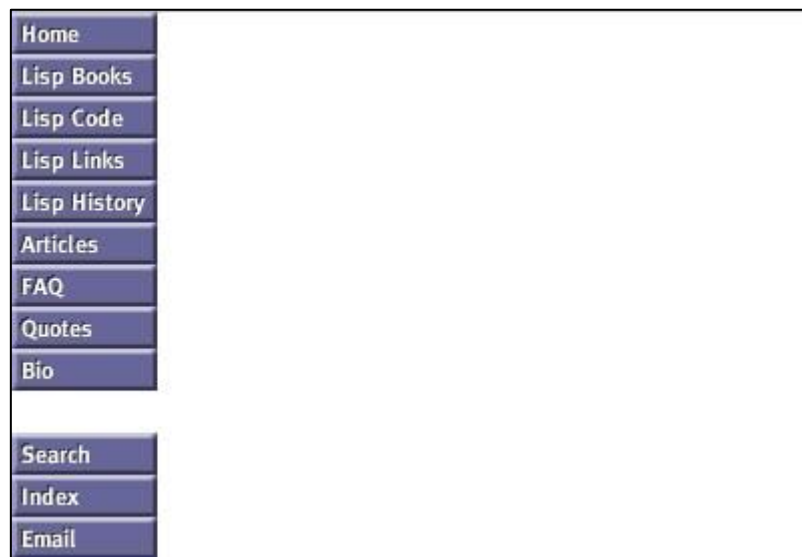
律速になるところまで最適化した。ユーザのデータのサイズを減らすのは無理だから(簡単なことなんてないね)、そこで止められるということが分かったんだ。

Efficiency matters for server-based software, because you're paying for the hardware. The number of users you can support per server is the divisor of your capital cost, so if you can make your software very efficient you can undersell competitors and still make a profit. At Viaweb we got the capital cost per user down to about \$5. It would be less now, probably less than the cost of sending them the first month's bill. Hardware is free now, if your software is reasonably efficient.

効率化はサーバベースソフトウェアでは重要だ。ハードウェアに金を払っているのはあなたなのだから。サーバ当たりどれだけのユーザをサポートできるかが資本コストの分母に効いてくるから、ソフトウェアを非常に効率良く書けば、競争相手より安く提供してなお利益を上げることができる。Viaweb では、ユーザ当たりの資本コストを 5 ドル程度まで下げることができた。今ではもっと安いだろう。たぶん、最初の月の請求書を郵送するコストより安くなったんじゃないだろうか。ソフトウェアが十分に効率良ければ、今ではハードウェアはただみたいものだ。

Watching users can guide you in design as well as optimization. Viaweb had a scripting language called RTML that let advanced users define their own page styles. We found that RTML became a kind of suggestion box, because users only used it when the predefined page styles couldn't do what they wanted. Originally the editor put button bars across the page, for example, but after a number of users used RTML to put buttons down the left [side](#), we made that an option (in fact the default) in the predefined page styles.

ユーザを観察することは、最適化だけでなく、設計上の指針にもなる。Viaweb では、進んだユーザは RTML というスクリプティング言語を使って自分のページスタイルを定義できた。私たちは RTML が一種の目安箱になっていることに気付いた。というのも、ユーザが RTML を使うということは、組み込みのページスタイルではユーザのやりたいことができないということだったからだ。例えば、最初のエディタはボタンバーをページに横長に付けていたのだが、たくさんのユーザが RTML を使ってボタンを左側に縦に並べるのを見て、そういうスタイルを組み込みのものにした(実際、それをデフォルトにしたんだ)。



Finally, by watching users you can often tell when they're in trouble. And since the customer is always right, that's a sign of something you need to fix. At Viaweb the key to getting users was the online test drive. It was not just a series of slides built by marketing people. In our test drive, users actually used the software. It took about five minutes, and at the end of it they had built a real, working store.

最後に、ユーザを観察することで、ユーザがどこで困っているかを知ることができる。そして、ユーザはいつでも正しいのだから、それは何かを直さなくちゃならないということだ。Viaweb では、ユーザを得るための鍵はオンラインの試用サイトだった。広報部門が作ったスライドなんかじゃない。試用サイトでは、ユーザは実際にソフトウェアを使うことができた。5 分ほどで、実際に動作するオンラインストアを構築することができたんだ。

The test drive was the way we got nearly all our new users. I think it will be the same for most Web-based applications. If users can get through a test drive successfully, they'll like the product. If they get confused or bored, they won't. So anything we could do to get more people through the test drive would increase our growth rate.

試用サイトこそが、ほとんど全てのユーザを獲得した方法だった。たいていの Web ベースアプリケーションはそうなんじゃないだろうか。ユーザが試用サイトを無事最後まで進めたら、ユーザはその製品を気に入るだろう。でも、もし途中で混乱したり飽きたりしたら、気に入らないだろう。だから少しでも多くの人々に試用サイトを最後まで使ってもらうことが、成長を加速することに直結していた。

I studied click trails of people taking the test drive and found that at a certain step

they would get confused and click on the browser's Back button. (If you try writing Web-based applications, you'll find that the Back button becomes one of your most interesting philosophical problems.) So I added a message at that point, telling users that they were nearly finished, and reminding them not to click on the Back button. Another great thing about Web-based software is that you get instant feedback from changes: the number of people completing the test drive rose immediately from 60% to 90%. And since the number of new users was a function of the number of completed test drives, our revenue growth increased by 50%, just from that change.

私は試用サイトでの人々のクリック履歴を調べて、あるステップで多くのユーザが混乱してブラウザの **Back** ボタンをクリックすることを見つけた。(Web ベースのアプリケーションを書いて見ると、**Back** ボタンは最も哲学的な問題であることが分かるだろう)。そこで私はそのステップに、もうほとんど最後まで来たので **Back** ボタンを押さないようユーザにお願いするメッセージを入れた。Web ベースソフトウェアのもうひとつの素晴らしい点は、フィードバックが直ちに得られることだ。試用サイトを最後まで使う人の割合が、**60%** から **90%** に跳ね上がったんだ。新規ユーザの数は試用サイトを最後まで使った人の数に依存するから、このたったひとつの変更で、私たちの利益は **50%** 増加したことになる。

Money

お金の話

In the early 1990s I read an article in which someone said that software was a subscription business. At first this seemed a very cynical statement. But later I realized that it reflects reality: software development is an ongoing process. I think it's cleaner if you openly charge subscription fees, instead of forcing people to keep buying and installing new versions so that they'll keep paying you. And fortunately, subscriptions are the natural way to bill for Web-based applications. 1990 年代の始めごろ、ソフトウェアはサブスクリプションビジネス(購読モデルのビジネス)になるだろうという記事を読んだ。最初はそれがずいぶんシニカルな意見に思えたものだ。だが、後に私はそれが現実を反映していることを知った。ソフトウェア開発は継続するプロセスだ。だから、人々に次々と新しいバージョンをインストールさせることで収益を得るよりも、はっきりと購読料という形で料金を設定した方が明解だろうと思う。うまく具合に、Web ベースアプリケーションではサブスクリプション形式が自然な課金法だ。

Hosting applications is an area where companies will play a role that is not likely to be filled by freeware. Hosting applications is a lot of stress, and has real

expenses. No one is going to want to do it for free.

アプリケーションのホスティングは、フリーウェアによって会社の活動の余地がなくなってしまう分野のひとつだろう。ホスティングはストレスの多い活動だし、実際の経費もかかる。無料ではやっていけない。

For companies, Web-based applications are an ideal source of revenue. Instead of starting each quarter with a blank slate, you have a recurring revenue stream. Because your software evolves gradually, you don't have to worry that a new model will flop; there never need be a new model, per se, and if you do something to the software that users hate, you'll know right away. You have no trouble with uncollectable bills; if someone won't pay you can just turn off the service. And there is no possibility of piracy.

会社にとっても、Web ベースアプリケーションは理想的な収益源だ。毎四半期を白紙で始める代わりに、定常的な収益が上がってくる。ソフトウェアは徐々に進化するから、新モデルが失敗することを心配する必要はない。いわゆる新モデルというのが必要になることはないわけだし、ユーザがソフトウェアの何かを気に入らなかつたらすぐにそれと知ることができる。回収できない請求を抱えることもない。料金を払わない人に対しては、サービスを打ち切るだけでいい。また、海賊版が出回る可能性もない。

That last "advantage" may turn out to be a problem. Some amount of piracy is to the advantage of software companies. If some user really would not have bought your software at any price, you haven't lost anything if he uses a pirated copy. In fact you gain, because he is one more user helping to make your software the standard-- or who might buy a copy later, when he graduates from high school.

ここに挙げた最後の「利点」は、もしかすると問題になるかもしれない。ソフトウェア会社にとっては、ある程度の海賊版は役に立つこともある。どんな値段でも正規には絶対に買わないというユーザがいるのなら、その人が海賊版を使ったところで何も失ってはいないわけだ。実際は、そういう人もあなたのソフトウェアをデファクトスタンダードにするのに役立つ一人になってくれているから、利益であるとも言える。それに、そういう人だって高校を卒業したら、ちゃんと金を払って正規版を買ってくれるかもしれない。

When they can, companies like to do something called price discrimination, which means charging each customer as much as they can afford. [8] Software is particularly suitable for price discrimination, because the marginal cost is close to zero. This is why some software costs more to run on Suns than on Intel boxes: a company that uses Suns is not interested in saving money and can safely be

charged more. Piracy is effectively the lowest tier of price discrimination. I think that software companies understand this and deliberately turn a blind eye to some kinds of piracy. [9] With server-based software they are going to have to come up with some other solution.

会社は可能な限り、価格差別と呼ばれるものをやりたがる。各ユーザに対してそれぞれが払えるだけ払わせるというやり方だ。[8] ソフトウェアは限界コストがゼロに近いので、特に価格差別に適している。これが、Sun 上で走るバージョンが Intel PC で走るバージョンより高いソフトウェアがある理由だ。Sun を使っている企業は金にしぶくないから、値段を高くしても大丈夫というわけだ。海賊版は、実質的に価格差別の最下層に当たるんじゃないか。ソフトウェア会社はこれを理解して、ある種の海賊版を黙認しているんじゃないかと思う。[9] サーバベースのソフトウェアでは、そういう会社は別のソリューションを必要とするだろう。

Web-based software sells well, especially in comparison to desktop software, because it's easy to buy. You might think that people decide to buy something, and then buy it, as two separate steps. That's what I thought before Viaweb, to the extent I thought about the question at all. In fact the second step can propagate back into the first: if something is hard to buy, people will change their mind about whether they wanted it. And vice versa: you'll sell more of something when it's easy to buy. I buy more books because Amazon exists. Web-based software is just about the easiest thing in the world to buy, especially if you have just done an online demo. Users should not have to do much more than enter a credit card number. (Make them do more at your peril.)

Web ベースソフトウェアは、特にデスクトップソフトウェアに比べ、よく売れるだろう。買うのが簡単だからだ。人がものを買う時に、まず買うことを決めて、それから買う、という 2 つのステップがあると思うかもしれない。私も Viaweb を始める前はそう思っていた。もっともほとんどそんなことを考えたことはなかったんだが。ところが実際は、第 2 のステップが最初のステップへと逆に伝搬するんだ。つまり、何かを買うことが難しい場合、人々は自分の心を変えて、それが欲しくないと思うようにするんだ。逆も成り立つ。買いやすいようになっていれば、より売れる。Amazon ができて私は本をたくさん買うようになった。Web ベースのソフトウェアは世界中で最も買いやすいもののひとつだろう。特にオンラインデモを終えたばかりの人にとっては。ユーザは、クレジットカード番号を入れる以上の作業をしなくて済むようになっていくべきだ(それ以上の作業をさせるなら覚悟しておくこと)。

Sometimes Web-based software is offered through ISPs acting as resellers. This is

a bad idea. You have to be administering the servers, because you need to be constantly improving both hardware and software. If you give up direct control of the servers, you give up most of the advantages of developing Web-based applications.

しばしば、Web ベースソフトウェアは ISP をリセラーとして提供される。これは悪いアイデアだと思う。自分でサーバを管理すべきだ。それによってハードウェアとソフトウェアの両方を常に改良してゆくことができる。サーバへの直接の制御を手放したら、Web ベースアプリケーションを開発する利点の多くを手放すことになる。

Several of our competitors shot themselves in the foot this way-- usually, I think, because they were overrun by suits who were excited about this huge potential channel, and didn't realize that it would ruin the product they hoped to sell through it. Selling Web-based software through ISPs is like selling sushi through vending machines.

私たちのライバル会社のいくつかは、この方法で自滅してしまった。たぶん彼らは、ISP 経由にすることで得られるかもしれない巨大な収入に目が眩んだスーツ族に乗っ取られて、それによって売るべき製品を台無しにしてしまうことに気付かなかったのだろう。Web ベースソフトウェアを ISP を通じて売るとするのは、寿司を自動販売機で売のようなものだ。

Customers

顧客

Who will the customers be? At Viaweb they were initially individuals and smaller companies, and I think this will be the rule with Web-based applications. These are the users who are ready to try new things, partly because they're more flexible, and partly because they want the lower costs of new technology.

どういった人々が顧客になるだろう。Viaweb では、最初は個人や小さな会社だった。私はそれが Web ベースアプリケーションの主要な顧客になるだろうと思う。そういったユーザは新しいことに挑戦してみたがる。考え方が柔軟だし、新しい技術によって経費を下げたいという要求もあるからだ。

Web-based applications will often be the best thing for big companies too (though they'll be slow to realize it). The best intranet is the Internet. If a company uses true Web-based applications, the software will work better, the servers will be better administered, and employees will have access to the system from anywhere.

Web ベースのアプリケーションは、大会社にとっても最良のものになり得る(ただ、大会社がそれに気付くのは遅いだろう)。最良のイントラネットはインターネットだ。会社が本当の Web ベースアプリケーションを使えば、ソフトウェアはよりうまく動き、サーバはもっと良く管理され、従業員はどこからでもシステムにアクセスできる。

The argument against this approach usually hinges on security: if access is easier for employees, it will be for bad guys too. Some larger merchants were reluctant to use Viaweb because they thought customers' credit card information would be safer on their own servers. It was not easy to make this point diplomatically, but in fact the data was almost certainly safer in our hands than theirs. Who can hire better people to manage security, a technology startup whose whole business is running servers, or a clothing retailer? Not only did we have better people worrying about security, we worried more about it. If someone broke into the clothing retailer's servers, it would affect at most one merchant, could probably be hushed up, and in the worst case might get one person fired. If someone broke into ours, it could affect thousands of merchants, would probably end up as news on CNet, and could put us out of business.

それに対する反論は、セキュリティに関するものだ。もし従業員が簡単にアクセスできるなら、悪い人間も簡単にアクセスできてしまうんじゃないか。いくつかの大手の商店は、クレジットカード情報は自社サーバにあったほうが安全だという理由で、Viaweb を使うのをためらっていた。この点を角が立たないように説明するのは難しいのだが、現実的にはほぼ確実に、データは彼らに任せておくより私たちが扱った方が安全だ。サーバ運営に特化した技術ベンチャーと、衣料小売店と、どっちがセキュリティを管理するのによい人材を雇えるだろうか。しかも、ただセキュリティの面倒を見る適任者を雇っていたというだけじゃない。私たちは彼らよりもっとセキュリティに神経を尖らせていた。もし誰かが衣料小売店のサーバに侵入したとしても、それはせいぜいその小売店にしか影響を与えないし、大したニュースにもならず、悪くても責任者一人が首になるだけだろう。だが、もし私たちのサーバが侵入されたら、それは何千という店に影響をあたえ、たぶん CNet あたりにニュースが出て、私たちは潰れていたかもしれない。

If you want to keep your money safe, do you keep it under your mattress at home, or put it in a bank? This argument applies to every aspect of server administration: not just security, but uptime, bandwidth, load management, backups, etc. Our existence depended on doing these things right. Server problems were the big no-no for us, like a dangerous toy would be for a toy maker, or a salmonella outbreak for a food processor.

金を安全に保管したかったら、ベッドの下に隠しておくのと銀行に預けるのと、どっちを選べばいいだろうか?この議論はサーバ管理の全ての面に適用できる。セキュリティだけじゃない。アップタイム、ネットワーク帯域、負荷管理、バックアップ、等々。私たちは、これらの仕事をちゃんとこなすことを前提に存在しているんだ。サーバが問題を起こすのは、私たちにとって何が何でも避けるべきことだった。ちょうど、おもちゃ屋にとっての危険なおもちゃとか、食品業者にとってのサルモネラ菌などと同じようなものだ。

A big company that uses Web-based applications is to that extent outsourcing IT. Drastic as it sounds, I think this is generally a good idea. Companies are likely to get better service this way than they would from in-house system administrators. System administrators can become cranky and unresponsive because they're not directly exposed to competitive pressure: a salesman has to deal with customers, and a developer has to deal with competitors' software, but a system administrator, like an old bachelor, has few external forces to keep him in line. [\[10\]](#) At Viaweb we had external forces in plenty to keep us in line. The people calling us were customers, not just co-workers. If a server got wedged, we jumped; just thinking about it gives me a jolt of adrenaline, years later.

大会社にとって、Web ベースアプリケーションを使うことは IT をアウトソースするということになる。過激に聞こえるかもしれないが、私はそれは一般的に良いことだと思う。社内のシステム管理者に任せるよりも良いサービスを得られるようになる会社が多いだろう。システム管理者は競争的なプレッシャーに晒されていないと、融通がきかなくなり、適切な反応ができなくなる。セールスマンは常に顧客に接しているし、開発者はライバルのソフトウェアと競わなくちゃならない。でもシステム管理者は、独身の老人のように、きちんとしなくちゃならないっていう外圧があまりないんだ。 [\[10\]](#) Viaweb では、私たちはきちんとしなくちゃならないって外圧がうんざりするほどあった。何かあつて電話をかけてくるのは同僚じゃない、顧客だ。サーバがおかしくなりそうな気配を見せるだけで私たちは飛び上がったものだ。何年もたった今でさえ、そのことを考えるだけでアドレナリンがどっと出るのを感じる。

So Web-based applications will ordinarily be the right answer for big companies too. They will be the last to realize it, however, just as they were with desktop computers. And partly for the same reason: it will be worth a lot of money to convince big companies that they need something more expensive.

だから Web ベースのアプリケーションは、大会社にとっても普通に正しいソリューションとなってゆくだろう。ただ、大会社は誰よりも遅くそのことに気付くだろう。ちょうどデスクトップコンピュータに関して彼らがそうだったように。たぶんその理由も似たような

ものだ。大会社により高いものを売りつけるためにより多くの金を費すことには、十分に見返りがあるからだ。

There is always a tendency for rich customers to buy expensive solutions, even when cheap solutions are better, because the people offering expensive solutions can spend more to sell them. At Viaweb we were always up against this. We lost several high-end merchants to Web consulting firms who convinced them they'd be better off if they paid half a million dollars for a custom-made online store on their own server. They were, as a rule, not better off, as more than one discovered when Christmas shopping season came around and loads rose on their server. Viaweb was a lot more sophisticated than what most of these merchants got, but we couldn't afford to tell them. At \$300 a month, we couldn't afford to send a team of well-dressed and authoritative-sounding people to make presentations to customers.

金持ちの顧客は、たとえ安くても良いソリューションがあっても、高いソリューションを買う傾向がある。高いソリューションを提供するところはそれを売るためにも多くの費用をかけているからだ。Viaweb ではいつもそういうことに反対していた。50 万ドルで自社サーバ上にカスタムオンラインストアを作ることを提案した Web コンサルティングファームに、ハイエンドの顧客をいくつか持っていかれた。別に彼らが私たちより良かったわけじゃない。クリスマス商戦が近づいて、サーバロードが跳ね上がった時に、そのことは露呈した。Viaweb はそういう大口顧客が建てたサーバよりもずっと優れたシステムを構築していたのだけれど、それを伝えるだけの予算がなかったのだ。月に 300 ドルという広告経費では、さっぱりとしたスーツをびしっと着こなして荘重なプレゼンを披露する営業部隊を雇うことなんてとてもできなかった。

A large part of what big companies pay extra for is the cost of selling expensive things to them. (If the Defense Department pays a thousand dollars for toilet seats, it's partly because it costs a lot to sell toilet seats for a thousand dollars.) And this is one reason intranet software will continue to thrive, even though it is probably a bad idea. It's simply more expensive. There is nothing you can do about this conundrum, so the best plan is to go for the smaller customers first. The rest will come in time.

大企業が余分に支払っている金の多くは、それをその会社売り込むために使われている。(例えば、国防省がトイレの便座に 1000 ドル払っているとしよう。それがそんなに高価なのは、トイレの便座を 1000 ドルで売るには多大な経費がかかるからだ)。この原理のおかげで、イントラネットソフトウェアは、おそらく悪いアイディアであるにもかかわらず、

長く生き残るだろう。単にそれが高価だというだけで。この難問に対してできることはほとんどない。だから、最初に小さな顧客を相手にするのがよいだろう。残りは後からついてくる。

Son of Server

サーバの息子

Running software on the server is nothing new. In fact it's the old model: mainframe applications are all server-based. If server-based software is such a good idea, why did it lose last time? Why did desktop computers eclipse mainframes?

ソフトウェアをサーバで走らせるのは別に新しいことではない。いや、むしろ古いモデルと言えるだろう。メインフレームの時代、アプリケーションは全てサーバベースだった。サーバベースソフトウェアがそんなに良いアイディアなんだとしたら、どうして前回、このモデルは敗れたんだろうか。なぜ、デスクトップコンピュータがメインフレームに勝ったんだろう。

At first desktop computers didn't look like much of a threat. The first users were all hackers-- or hobbyists, as they were called then. They liked microcomputers because they were cheap. For the first time, you could have your own computer. The phrase "personal computer" is part of the language now, but when it was first used it had a deliberately audacious sound, like the phrase "personal satellite" would today.

当初、デスクトップコンピュータは大した脅威には見えなかった。最初のユーザたちはみんなハッカーだった(当時はホビイストと呼ばれていた)。彼らがマイクロコンピュータを気に入ったのは、それが安かったからだ。初めて、彼らは自分のコンピュータというものを所有することができたんだ。「パーソナルコンピュータ」というフレーズは今では普通の言葉だが、最初に使われた時、それにはむしろ自慢げな響きが含まれていた。ちょうど、現代で「パーソナルサテライト」とかいう言葉の響きのようなものだ。

Why did desktop computers take over? I think it was because they had better software. And I think the reason microcomputer software was better was that it could be written by small companies.

どうしてデスクトップコンピュータが台頭したんだろう。私の考えは、デスクトップのほうにより良いソフトウェアがあったからだ。そしてマイクロコンピュータのソフトウェアのほうが良かった理由は、小さな会社がそれを書くことが可能だったからだ。

I don't think many people realize how fragile and tentative startups are in the earliest stage. Many startups begin almost by accident-- as a couple guys, either with day jobs or in school, writing a prototype of something that might, if it looks promising, turn into a company. At this larval stage, any significant obstacle will stop the startup dead in its tracks. Writing mainframe software required too much commitment up front. Development machines were expensive, and because the customers would be big companies, you'd need an impressive-looking sales force to sell it to them. Starting a startup to write mainframe software would be a much more serious undertaking than just hacking something together on your Apple II in the evenings. And so you didn't get a lot of startups writing mainframe applications.

立ち上がったばかりのベンチャー企業が、どれだけ脆く、不確かなものであるかを知っている人はあまりいないんじゃないかと思う。多くのベンチャー企業は、ほとんど偶然のようにして始まる。例えば、昼は仕事を持っていたり学校に行っていたりする 2 人の若者が、ちょろっとプロトタイプを書いてみて、それがいけそうだったら会社にするってな具合だ。この幼生期においては、適当な大きさの障害がひとつあれば会社はそこで止まってしまう。メインフレームソフトウェアを書くためには、始めるまえに多大なコミットメントが必要だ。開発機器は高価だし、顧客は大会社になるから、よりすぐりのセールス部隊を用意して売り込みをかけなくちゃならない。メインフレームソフトウェアを書くベンチャーを立ち上げるのは、夜に自分の Apple II をハックして何かを書くようなベンチャーよりもずっと大がかりな事業になる。これが、メインフレームアプリケーションを書くベンチャーがあまり生まれない理由だ。

The arrival of desktop computers inspired a lot of new software, because writing applications for them seemed an attainable goal to larval startups. Development was cheap, and the customers would be individual people that you could reach through computer stores or even by mail-order.

デスクトップコンピュータの到来は多くの新しいソフトウェアを生み出した。幼生期のベンチャーにとって、それ用のソフトを書くことは到達可能なゴールに見えたからだ。開発は安価で、顧客はコンピュータショップや手紙での注文をしてくるような個人だった。

The application that pushed desktop computers out into the mainstream was [VisiCalc](#), the first spreadsheet. It was written by two guys working in an attic, and yet did things no mainframe software could do. [\[11\]](#) VisiCalc was such an advance, in its time, that people bought Apple IIs just to run it. And this was the

beginning of a trend: desktop computers won because startups wrote software for them.

デスクトップコンピュータを主流にのし上げたアプリケーションは、最初の表計算ソフト、VisiCalc だ。それは 2 人の若者によって屋根裏で書かれ、それでいてどのメインフレームソフトウェアもなし得なかった機能を持っていた。[\[11\]](#) VisiCalc が当時はあまりに斬新だったため、人々はそれを走らせるためだけに Apple II を買ったものだ。そして、これが現在に至る流れの始まりだった。ベンチャーがソフトウェアを書くことで、デスクトップコンピュータが勝利を収める。

It looks as if server-based software will be good this time around, because startups will write it. Computers are so cheap now that you can get started, as we did, using a desktop computer as a server. Inexpensive processors have eaten the workstation market (you rarely even hear the word now) and are most of the way through the server market; Yahoo's servers, which deal with loads as high as any on the Internet, all have the same inexpensive Intel processors that you have in your desktop machine. And once you've written the software, all you need to sell it is a Web site. Nearly all our users came direct to our site through word of mouth and references in the press. [\[12\]](#)

とすれば、現代においてはサーバベースソフトウェアが有利になるように思える。ベンチャー企業がそれを書くだろうから。コンピュータの価格がこれだけ下がったため、私たちがやったように、誰もがデスクトップコンピュータをサーバとして使って始めることができる。安価なプロセッサはワークステーションの市場をすっかり喰い尽くし(今では「ワークステーション」という言葉を聞くことさえ滅多になくなった)、そしてサーバの市場にも大きく食い込んでいる。インターネット上でおそらく最も多くの負荷をさばいているサーバのひとつであろう Yahoo! のサーバは、デスクトップマシンが使っているのと同じ、安価な Intel プロセッサを使っている。そしてソフトウェアを書き上げたら、それを売るのに必要なのは Web サイトひとつだ。Viaweb では、ユーザのほとんどはロコミか報道を見て直接我々のサイトを訪れた。[\[12\]](#)

Viaweb was a typical larval startup. We were terrified of starting a company, and for the first few months comforted ourselves by treating the whole thing as an experiment that we might call off at any moment. Fortunately, there were few obstacles except technical ones. While we were writing the software, our Web server was the same desktop machine we used for development, connected to the outside world by a dialup line. Our only expenses in that phase were food and rent.

Viaweb は典型的な幼生期のベンチャー企業だった。会社を始めるってこと自体ひどくおっかないことだったから、最初の数ヵ月は、全てはひとつの実験で、止めたくなったらいつでも止められると自分たちに言い聞かせていたほどだ。幸いなことに、技術的な問題以外で邪魔になるものはほとんどなかった。ソフトウェアを書いている間、私たちの Web サーバは開発に使っていたデスクトップマシンと同じもので、ダイヤルアップ回線を通じて外の世界につながっていた。この時期の経費は食費と家賃だけだった。

There is all the more reason for startups to write Web-based software now, because writing desktop software has become a lot less fun. If you want to write desktop software now you do it on Microsoft's terms, calling their APIs and working around their buggy OS. And if you manage to write something that takes off, you may find that you were merely doing market research for Microsoft.

いまや、ベンチャー企業が Web ベースソフトウェアを書くべき理由はもつとずっとたくさんある。デスクトップソフトウェアを書くのはどんどんつまらなくなっているからだ。デスクトップソフトウェアを書こうと思ったら、マイクロソフトの土俵にあがらなくちゃならない。OS のバグを回避しつつ、彼らの API を呼ばなくちゃならない。そしてやっこのことで、売れるソフトウェアを書き上げた時、あなたは単にマイクロソフトのマーケットリサーチをやっただけだったということを知るのだ。

If a company wants to make a platform that startups will build on, they have to make it something that hackers themselves will want to use. That means it has to be inexpensive and well-designed. The Mac was popular with hackers when it first came out, and a lot of them wrote software for it. [13] You see this less with Windows, because hackers don't use it. The kind of people who are good at writing software tend to be running Linux or FreeBSD now.

ベンチャーに何かを書きたいと思わせるプラットフォームを作ろうとするなら、それはハッカーたちが使ってみたいと思うようなものじゃなくちゃいけない。つまり、安価で、しかも設計が良いものだ。Mac は最初に出た当時はハッカーの間でとても人気があり、たくさんハッカーたちがソフトウェアを書いたものだ。[13] Windows ではそういうことがあまり起こっていない。ハッカーたちが使っていないからだ。ソフトウェアを書くのが得意な人は、今では Linux や FreeBSD を使っているだろう。

I don't think we would have started a startup to write desktop software, because desktop software has to run on Windows, and before we could write software for Windows we'd have to use it. The Web let us do an end-run around Windows, and deliver software running on Unix direct to users through the browser. That is a

liberating prospect, a lot like the arrival of PCs twenty-five years ago.

私たちは、たとえチャンスがあっても、デスクトップソフトウェアを書く会社を立ち上げることはなかっただろうと思う。売れるデスクトップソフトウェアは **Windows** 上で走らなくちゃならない。ということは、ソフトウェアを書く以前に **Windows** を使わなくちゃならないということだ。Web を利用することで **Windows** を回避することができたし、ブラウザを通じてユーザに直接 **Unix** 上で走るソフトウェアを届けることができた。これはとても解放的な見通しだった。ちょうど、25 年前の PC の到来がそうであったように。

Microsoft

マイクロソフト

Back when desktop computers arrived, IBM was the giant that everyone was afraid of. It's hard to imagine now, but I remember the feeling very well. Now the frightening giant is Microsoft, and I don't think they are as blind to the threat facing them as IBM was. After all, Microsoft deliberately built their business in IBM's blind spot.

デスクトップコンピュータが到来した当時、**IBM** は誰もが恐れる巨人だった。今それを想像するのは難しいかもしれないが、私はあの感じをまだ覚えている。現在、その恐るべき巨人はマイクロソフトだ。そして、彼らは、かつての **IBM** のように迫り来る脅威に鈍感ではないと思う。実際、マイクロソフトは意識的に **IBM** の死角を突いて成り上がってきたのだから。

I mentioned earlier that my mother doesn't really need a desktop computer. Most users probably don't. That's a problem for Microsoft, and they know it. If applications run on remote servers, no one needs Windows. What will Microsoft do? Will they be able to use their control of the desktop to prevent, or constrain, this new generation of software?

私は最初の方で、私の母はデスクトップコンピュータなんて必要じゃないんだと述べた。たぶんほとんどのユーザがそうだろう。これがマイクロソフトの問題であり、そして彼らはそれに気付いている。アプリケーションが離れたサーバで動くなら、誰も **Windows** を必要としない。ではマイクロソフトはどうするだろうか。デスクトップ上で持っている力を使って、この新世代のソフトウェアを邪魔したり、制限したりすることが彼らにできるだろうか。

My guess is that Microsoft will develop some kind of server/desktop hybrid, where the operating system works together with servers they control. At a minimum,

files will be centrally available for users who want that. I don't expect Microsoft to go all the way to the extreme of doing the computations on the server, with only a browser for a client, if they can avoid it. If you only need a browser for a client, you don't need Microsoft on the client, and if Microsoft doesn't control the client, they can't push users towards their server-based applications.

私の推測は、マイクロソフトはある種のサーバ/デスクトップ複合体を開発するというものだ。OS は彼らが制御するサーバと連携して動作する。少なくとも、ファイルに関してはユーザが望めば中央のサーバに置いておけるようになるだろう。ただ、マイクロソフトが、クライアントにブラウザだけを置いて残りの全ての計算をサーバまで持ってゆくまでやるとは私は思わない。むしろ可能な限りそれは避けるだろう。なぜってクライアント側にブラウザだけが必要なら、クライアントにマイクロソフトは必要ないってことだから。クライアントで制御を失えば、ユーザを彼らのサーバベースアプリケーションへと引っ張り込むこともできなくなる。

I think Microsoft will have a hard time keeping the genie in the bottle. There will be too many different types of clients for them to control them all. And if Microsoft's applications only work with some clients, competitors will be able to trump them by offering applications that work from any client. [\[14\]](#)

マイクロソフトは、このランプの妖精をランプに閉じ込めておくのに苦勞するだろう。いろいろなタイプのクライアントが出てくれば、それを全てコントロールするのは非常に難しくなる。そしてマイクロソフトのアプリケーションが特定のタイプのクライアントでしか動作しないとなれば、ライバルは全てのクライアントで動作するアプリケーションを出すことでマイクロソフトに勝てるだろう。 [\[14\]](#)

In a world of Web-based applications, there is no automatic place for Microsoft. They may succeed in making themselves a place, but I don't think they'll dominate this new world as they did the world of desktop applications.

Web ベースアプリケーションの世界には、マイクロソフトの場所はあらかじめ用意されてはいない。もちろん彼らは自分の領地をある程度確保することに成功するかもしれないが、デスクトップアプリケーションの世界を制覇したようにこの新しい世界を制覇することはできないだろうと思う。

It's not so much that a competitor will trip them up as that they will trip over themselves. With the rise of Web-based software, they will be facing not just technical problems but their own wishful thinking. What they need to do is cannibalize their existing business, and I can't see them facing that. The same

single-mindedness that has brought them this far will now be working against them. IBM was in exactly the same situation, and they could not master it. IBM made a late and half-hearted entry into the microcomputer business because they were ambivalent about threatening their cash cow, mainframe computing. Microsoft will likewise be hampered by wanting to save the desktop. A cash cow can be a damned heavy monkey on your back.

ライバルに食われるだけでなく、自分自身に食われることも心配しなくてはならない。**Web** ベースソフトウェアの台頭により、彼らは単なる技術的な問題だけではなく、自分の願望というやつに向き合わなくちゃならなくなるだろう。本当に必要なのは、彼らが今持っているビジネスの基盤を自ら崩すことなんだが、まだその決断ができるとは思えない。彼らをここまで発展させてきた強い思い込みが、ここに来て邪魔になっている。**IBM** もかつて全く同じ状況に置かれ、それを完全に乗り越えることはできなかった。メインフレームという金のなる木を傷つけることを恐れるあまり、**IBM** のマイクロコンピュータビジネスへの参入は、遅く、中途半端なものとなった。マイクロソフトも同じように、デスクトップを救おうと望むあまりに動きがとれなくなるかもしれない。金のなる木は、それを背負っていかなくちゃならないとしたら、ものすごく重い荷物になる。

I'm not saying that no one will dominate server-based applications. Someone probably will eventually. But I think that there will be a good long period of cheerful chaos, just as there was in the early days of microcomputers. That was a good time for startups. Lots of small companies flourished, and did it by making cool things.

サーバベースアプリケーションの世界に支配者が現れないと言っているわけじゃないよ。いずれ誰かがこの世界を支配するだろう。でもたぶんそうなるまでに、かなり長い、陽気などんちゃん騒ぎが続くと思う。マイクロコンピュータの出現期がそうであったように。それはベンチャー企業にとって良い時期だった。たくさんの小さな会社が芽を出し、クールなものを作ることで花を咲かせた。

Startups but More So

スタートアップ

The classic startup is fast and informal, with few people and little money. Those few people work very hard, and technology magnifies the effect of the decisions they make. If they win, they win big.

伝統的なベンチャー企業は、少人数で資本も少ないが、動きが速く、格式ばっていない。人数は少ないが猛烈に働く。技術は彼らの決断の効果を拡大する。勝てば見返りは大きい。

In a startup writing Web-based applications, everything you associate with startups is taken to an extreme. You can write and launch a product with even fewer people and even less money. You have to be even faster, and you can get away with being more informal. You can literally launch your product as three guys sitting in the living room of an apartment, and a server collocated at an ISP. We did.

Web ベースアプリケーションを書くベンチャー企業は、あなたがベンチャー企業と聞いて思い浮かべる全ての要素を最大限に引き伸ばしたようなところだ。より少ない人数と資本で製品を書いて公開できる。より速く動かなくちゃならないし、社内はよりくだけた感じになる。なにしろ、製品を公開するのに、マンションの一室のリビングルームに 3 人で座ったままできるんだから。サーバは ISP に置いておけばいい。それが私たちのやったことだ。

Over time the teams have gotten smaller, faster, and more informal. In 1960, software development meant a roomful of men with horn rimmed glasses and narrow black neckties, industriously writing ten lines of code a day on IBM coding forms. In 1980, it was a team of eight to ten people wearing jeans to the office and typing into vt100s. Now it's a couple of guys sitting in a living room with laptops. (And jeans turn out not to be the last word in informality.)

時代を経るにつれ、チームはより小さく、より速く、そしてよりくだけたものになってきた。1960 年には、ソフトウェア開発というものは、鼈甲縁の眼鏡をかけ細い黒ネクタイを締めた部屋いっぱいの男たちが、IBM のコーディングシートを一日に 10 行づつ黙々と埋めて行くというものだった。1980 年には、ジーンズを穿いて出勤する 8~10 人のチームが、VT100 に向かってプログラムをタイプしていた。今や、リビングで 2 人のプログラマがラップトップに向かってプログラムするようになっている(それに、ジーンズはまだ十分にカジュアルじゃないってことも証明された)。

Startups are stressful, and this, unfortunately, is also taken to an extreme with Web-based applications. Many software companies, especially at the beginning, have periods where the developers slept under their desks and so on. The alarming thing about Web-based software is that there is nothing to prevent this becoming the default. The stories about sleeping under desks usually end: then at last we shipped it and we all went home and slept for a week. Web-based software never ships. You can work 16-hour days for as long as you want to. And because you can, and your competitors can, you tend to be forced to. You can, so you must.

It's Parkinson's Law running in reverse.

ベンチャー企業はプレッシャーも大きいですが、この点に関しても、残念ながら Web ベースアプリケーションでは最大限になる。多くのソフトウェア会社では、とりわけ初めのうちは、開発者が机の下で眠らなければならないような期間がある。Web ベースソフトウェアで恐いのは、それが日常になることへの歯止めがないことだ。普通の「机の下で眠る」話には必ず終りがある。ソフトウェアをリリースして、みんな家に帰って 1 週間ぐっすり眠るんだ。だが、Web ベースソフトウェアはリリースしておしまいというものじゃない。やろうと思えばいくらでも、一日 16 時間働き続けられる。しかも、あなたがそれをできるし、ライバルもそれができるということは、結果としてそうせざるを得なくなりがちだってことだ。可能なら、やらねばならない。これはパーキンソンの法則を逆に適用しているようなものだ。

The worst thing is not the hours but the responsibility. Programmers and system administrators traditionally each have their own separate worries. Programmers have to worry about bugs, and system administrators have to worry about infrastructure. Programmers may spend a long day up to their elbows in source code, but at some point they get to go home and forget about it. System administrators never quite leave the job behind, but when they do get paged at 4:00 AM, they don't usually have to do anything very complicated. With Web-based applications, these two kinds of stress get combined. The programmers become system administrators, but without the sharply defined limits that ordinarily make the job bearable.

もっと悪いのは、労働時間ではなく、責任だ。プログラマとシステム管理者は、普通はそれぞれ違った心配を抱えているものだ。プログラマはバグについて心配するし、システム管理者はインフラについて心配する。プログラマは長時間、肘までどっぷりソースに漬かって仕事をしているが、あるところまで到達したら、家に帰ってそれを忘れることができる。システム管理者は仕事のことをすっかり忘れてしまうことはできないが、朝 4 時に呼び出しを食らった場合でも、やるべき作業はたいして複雑でない場合が多い。Web ベースアプリケーションでは、このふたつの異なった責任が一緒にかかってくるんだ。プログラマはシステム管理者になり、普通なら仕事を耐えられるものにしてくれるはずだった明確な責任の限界がなくなる。

At Viaweb we spent the first six months just writing software. We worked the usual long hours of an early startup. In a desktop software company, this would have been the part where we were working hard, but it felt like a vacation compared to the next phase, when we took users onto our server. The second

biggest benefit of selling Viaweb to Yahoo (after the money) was to be able to dump ultimate responsibility for the whole thing onto the shoulders of a big company.

Viaweb では最初の 6 ヶ月かそこいらは、ソフトウェアを書くだけで過ごしていた。普通のベンチャー企業の立ち上げ時と同じように、私たちは長時間働いていた。デスクトップソフトウェア会社では、そういう時期が一番きつい時期ということになる。しかし、我々が経験した次の時期、ユーザをサーバに迎えた時期に比べたら、ソフトを書くだけで良かった最初の期間はまるで休暇みたいなものだった。Viaweb を Yahoo! に売却したことによる 2 番目のメリットは(1 番目はもちろん金だが)、この究極の責任を全て大企業に背負ってもらえたということだろう。

Desktop software forces users to become system administrators. Web-based software forces programmers to. There is less stress in total, but more for the programmers. That's not necessarily bad news. If you're a startup competing with a big company, it's good news. [15] Web-based applications offer a straightforward way to outwork your competitors. No startup asks for more.

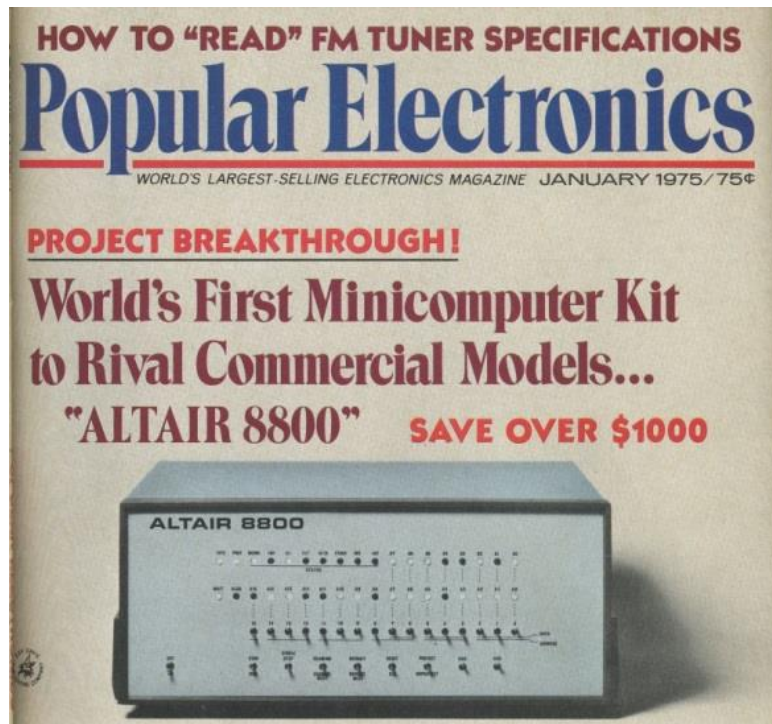
デスクトップソフトウェアはユーザがシステム管理者になることを強いるが、Web ベースソフトウェアではプログラマがシステム管理者になることを強いるとも言える。トータルとしてのストレスは減少するが、プログラマにとってのストレスは増大する。もともと、これは悪いことばかりではない。もしあなたが大会社と競っている立ち上がったばかりのベンチャーだとしたら、これは良いニュースだ。[15] Web ベースアプリケーションは競争相手に仕事で勝つための、直接の方法を提供してくれる。それはベンチャーにとって一番ありがたいものだ。

Just Good Enough

とりあえず使える

One thing that might deter you from writing Web-based applications is the lameness of Web pages as a UI. That is a problem, I admit. There were a few things we would have *really* liked to add to HTML and HTTP. What matters, though, is that Web pages are just good enough.

Web ベースアプリケーションを書くことをためらわせる要因のひとつは、Web ページを UI として見た場合の貧弱さだろう。これは確かに問題だ。HTML と HTTP に追加してもらえたらと切望することは確かにいくつかある。だが、重要なことは、Web ページはとりあえずは使えるということだ。



Popular Electronics, 1975 年 1 月号(部分)

There is a parallel here with the first microcomputers. The processors in those machines weren't actually intended to be the CPUs of computers. They were designed to be used in things like traffic lights. But guys like Ed Roberts, who designed the [Altair](#), realized that they were just good enough. You could combine one of these chips with some memory (256 bytes in the first Altair), and front panel switches, and you'd have a working computer. Being able to have your own computer was so exciting that there were plenty of people who wanted to buy them, however limited.

ここにも初期のマイクロコンピュータと通じるものがある。それらのマシンのプロセッサは、本物のコンピュータの CPU を目指していたわけじゃなかった。むしろ、例えば信号機のようなものを制御する機器として設計されていたんだ。でも、Altair を設計したエド・ロバーツのような人間が、それはとりあえず使うには十分だということに気付いた。そういうチップと、いくばくかのメモリ(最初の Altair では 256 バイトだった)を組み合わせ、スイッチ付きのフロントパネルをくっつければ、動作するコンピュータができた。自分でコンピュータを持てるということは当時はすごいことだったから、やれることにどんなに制限があっても、それを買いたいという人はたくさんいたんだ。

Web pages weren't designed to be a UI for applications, but they're just good

enough. And for a significant number of users, software that you can use from any browser will be enough of a win in itself to outweigh any awkwardness in the UI. Maybe you can't write the best-looking spreadsheet using HTML, but you can write a spreadsheet that several people can use simultaneously from different locations without special client software, or that can incorporate live data feeds, or that can page you when certain conditions are triggered. More importantly, you can write new kinds of applications that don't even have names yet. VisiCalc was not merely a microcomputer version of a mainframe application, after all-- it was a new type of application.

Web ページはもちろんアプリケーションの UI になるために設計されたわけじゃないけれど、それはとりあえずは使える。そして、非常に多くのユーザにとっては、どのブラウザでも使えるソフトウェアというのはそれだけで UI の不格好を補って余りあるメリットがある。もちろん、HTML でカッコいい表計算ソフトを書くことはできないだろう。でも、特別のクライアントをインストールすることなしに別々の場所から複数人で使える表計算ソフトがあったらどうだい?あるいはライブデータが取り込まれたり、何かの条件が満たされたら自動的に通知してくれる機能は?さらに重要なことに、まだ名前が付いてないような新しい種類のアプリケーションを書くことだってできる。VisiCalc はメインフレームアプリケーションの単なるマイコン版ではなかった。それは新しい種類のアプリケーションだったんだ。

Of course, server-based applications don't have to be Web-based. You could have some other kind of client. But I'm pretty sure that's a bad idea. It would be very convenient if you could assume that everyone would install your client-- so convenient that you could easily convince yourself that they all would-- but if they don't, you're hosed. Because Web-based software assumes nothing about the client, it will work anywhere the Web works. That's a big advantage already, and the advantage will grow as new Web devices proliferate. Users will like you because your software just works, and your life will be easier because you won't have to tweak it for every new client. [\[16\]](#)

もちろん、サーバベースのアプリケーションが必ずしも Web ベースでなければならないわけじゃない。他の種類のクライアントを使うことも考えられる。ただ、私はそれは悪いアイデアだと思う。皆があなただけのクライアントをインストールしてくれると仮定できるなら、こんなに便利なことはない。(あまりに便利だから、思わずそれができると信じてしまいそうだ)。でも、もし皆がそうしてくれなかったら、おしまいだ。Web ベースソフトウェアはクライアントに関して何も仮定しないために、Web が動作するところではどこでも動作する。これは既に大きな利点だ。新しい Web デバイスが出てくれば、その利点はさらに

増加する。ユーザは、何もしないでもあなたのソフトウェアが動くことから、それを気に入るだろうし、あなたは、新しいクライアントが出るたびにソフトウェアをいじらなくてよいから楽ができる。[16]

I feel like I've watched the evolution of the Web as closely as anyone, and I can't predict what's going to happen with clients. Convergence is probably coming, but where? I can't pick a winner. One thing I can predict is conflict between AOL and Microsoft. Whatever Microsoft's .NET turns out to be, it will probably involve connecting the desktop to servers. Unless AOL fights back, they will either be pushed aside or turned into a pipe between Microsoft client and server software. If Microsoft and AOL get into a client war, the only thing sure to work on both will be browsing the Web, meaning Web-based applications will be the only kind that work everywhere.

私は **Web** の進化を誰よりも密接に見てきた人間の一人のような気がしているが、それでもクライアント側で次に何が起こるかは予測できない。どこかに収束してゆくだろうが、それはどこだろう。誰が勝つかもわからない。一つ予測できるのは、**AOL** とマイクロソフトの衝突だ。マイクロソフトの **.NET** がどういう形になるにせよ、それはデスクトップとサーバをつなぐものを含むだろう。**AOL** は反撃しない限り、脇に押し退けられるか、単にマイクロソフトのクライアントとサーバソフトウェアをつなぐパイプに成り下がってしまう。マイクロソフトと **AOL** がクライアントの領域で戦うならば、動作を確実に保証できることは **Web** をブラウズすることだけになるから、**Web** ベースアプリケーションのみが、どこでも動作するという解になる。

How will it all play out? I don't know. And you don't have to know if you bet on Web-based applications. No one can break that without breaking browsing. The Web may not be the only way to deliver software, but it's one that works now and will continue to work for a long time. Web-based applications are cheap to develop, and easy for even the smallest startup to deliver. They're a lot of work, and of a particularly stressful kind, but that only makes the odds better for startups.

どういう結末が待っているだろう。私にはわからない。ただ、**Web** ベースのアプリケーションに賭けるのにこの結末を知る必要はないだろう。だって **Web** ベースアプリケーションを負かすには、ブラウザというモデルを壊さないとならない。**Web** はソフトウェアを届ける唯一の方法ではないけれど、現在も動作し、将来も長期にわたって動作しつづけるものになるだろう。**Web** ベースアプリケーションは安価に開発できて、最も小さなベンチャー企業だってそれを公開することができる。もちろん簡単な仕事じゃないし、とりわけストレスの高い仕事だが、だからこそそれはベンチャー企業にとって有利なんだ。

Why Not?

挑戦してみるかい？

E. B. White was amused to learn from a farmer friend that many electrified fences don't have any current running through them. The cows apparently learn to stay away from them, and after that you don't need the current. "Rise up, cows!" he wrote, "Take your liberty while despots snore!"

E.B.ホワイトは、農場を持っている友人から、電流を流せる柵の多くは実際には電流を切っていると聞いて感心したそう。牛は一度感電を経験したらそういう柵に近寄らないようになるから、電流を流し続ける必要はない。「立ち上がれ、牛たちよ！」彼は書いた。「君主が寝ている間に自由を手をせよ！」

If you're a hacker who has thought of one day starting a startup, there are probably two things keeping you from doing it. One is that you don't know anything about business. The other is that you're afraid of competition. Neither of these fences have any current in them.

あなたがハッカーで、いつかベンチャー企業を立ち上げたいと思っているとする。たぶん、2つの事柄があなたにそうすることをためらわせているんじゃないかと思う。ひとつは、あなたがビジネスについて何も知らないということ。もうひとつは、競争が怖いということ。実は、このどちらの柵にも電流は流れていないんだ。

There are only two things you have to know about business: build something users love, and make more than you spend. If you get these two right, you'll be ahead of most startups. You can figure out the rest as you go.

ビジネスに関しては、2つのことだけを知っておけばいい。ユーザが気に入るものを作ることと、使った金より多くの収入を得ることだ。この2つがちゃんと出来れば、それだけで多くのベンチャー企業よりも進んでいることになる。残りはやってゆくうちに分かってくるだろう。

You may not at first make more than you spend, but as long as the gap is closing fast enough you'll be ok. If you start out underfunded, it will at least encourage a habit of frugality. The less you spend, the easier it is to make more than you spend. Fortunately, it can be very cheap to launch a Web-based application. We launched on under \$10,000, and it would be even cheaper today. We had to spend thousands on a server, and thousands more to get SSL. (The only company selling

SSL software at the time was Netscape.) Now you can rent a much more powerful server, with SSL included, for less than we paid for bandwidth alone. You could launch a Web-based application now for less than the cost of a fancy office chair. 最初のうちは収入より支出が多くなるだろうけれど、その差額が十分に速く縮まっていていけば大丈夫だ。少ない資金で始めるなら、少なくとも節約の習慣はつけられる。支出が少なければ少ないほど、それを上回る収入を得ることは楽になる。幸い、Web ベースアプリケーションはとても安価に始められる。私たちは最初のサービスのスタートを 10,000 ドル以下でやったし、現在ならもっと安いだろう。数千ドルをサーバに、数千ドルを SSL を得るのに使った(当時、SSL ソフトウェアを売っていた唯一の会社はネットスケープだった)。現在では、SSL 付きでもっとずっと強力なサーバを、私たちがネットワーク帯域に払っていた料金よりも安く借りられるだろう。たぶん今なら、素敵なオフィスチェアよりも安く Web アプリケーションを立ち上げられるんじゃないだろうか。

As for building something users love, here are some general tips. Start by making something clean and simple that you would want to use yourself. Get a version 1.0 out fast, then continue to improve the software, listening closely to the users as you do. The customer is always right, but different customers are right about different things; the least sophisticated users show you what you need to simplify and clarify, and the most sophisticated tell you what features you need to add. The best thing software can be is easy, but the way to do this is to get the defaults right, not to limit users' choices. Don't get complacent if your competitors' software is lame; the standard to compare your software to is what it could be, not what your current competitors happen to have. Use your software yourself, all the time. Viaweb was supposed to be an online store builder, but we used it to make our own site too. Don't listen to marketing people or designers or product managers just because of their job titles. If they have good ideas, use them, but it's up to you to decide; software has to be designed by hackers who understand design, not designers who know a little about software. If you can't design software as well as implement it, don't start a startup.

ユーザが気に入るものという点に関しては、いくつか一般的なヒントがある。まず、自分でも使いたいと思うような、明快で簡単なものから作り始めることだ。バージョン 1.0 を素早く出して、それからユーザの声をよく聞きながらそれを改良してゆく。顧客は常に正しいというのは本当だけれど、それぞれの顧客は別々の点に関して正しいんだ。入門レベルのユーザは、どこを簡単で明快にすべきかを教えてくれる。最も進んだレベルのユーザは、どういう機能を足したら良いかを教えてくれる。最良のソフトウェアは、簡単であるものだが、それはデフォルト設定がそうになっているためで、ユーザが望めばその選択に

限界があつてはならない。ライバルのソフトウェアの出来が悪いからって悦に入つてはいけない。あなたのソフトウェアが比較すべきは、ライバルのソフトウェアの現状ではなく、あなたのソフトウェアのあるべき姿だ。自分で作ったソフトウェアを常に自分で使うこと。**Viaweb** はオンラインストアビルダーだったけれど、私たちは自分のサイトもそれで作っていた。マーケティングとかデザイナーとかプロダクトマネージャだとかいう人たちの意見を、単にそういう人たちがそういう肩書を持っているからというだけで鵜呑みにしないこと。彼らが良いアイディアを出してくれたら使えばいいけれど、それはあくまであなたの判断だ。ソフトウェアはデザインを知っているハッカーによってデザインされるべきで、ソフトウェアのことをほとんど知らないデザイナーによってデザインされるべきではない。ソフトウェアの実装はできるがデザインはできないというのなら、ベンチャーを立ち上げるのはあきらめたほうがいい。

Now let's talk about competition. What you're afraid of is not presumably groups of hackers like you, but actual companies, with offices and business plans and salesmen and so on, right? Well, they are more afraid of you than you are of them, and they're right. It's a lot easier for a couple of hackers to figure out how to rent office space or hire sales people than it is for a company of any size to get software written. I've been on both sides, and I know. When Viaweb was bought by Yahoo, I suddenly found myself working for a big company, and it was like trying to run through waist-deep water.

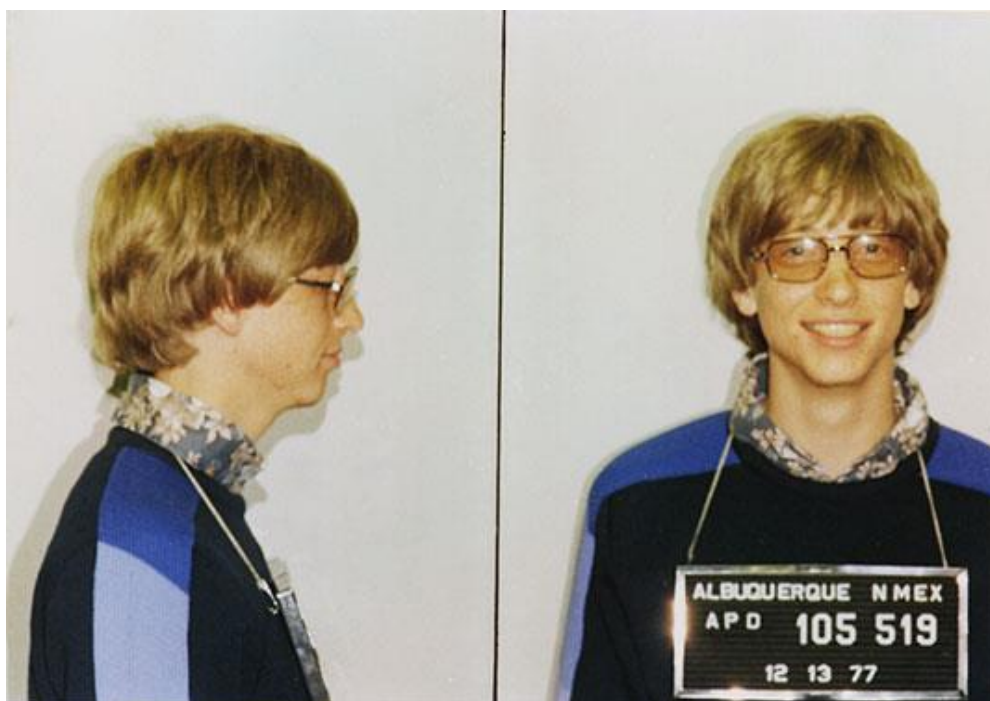
さて、競争について話そうか。たぶんあなたが恐れているのは、他のハッカーの集団ではなく、ちゃんとしたオフィスとビジネスプランとセールスマンとその他諸々を備えている本物の会社だろう。違うかい？本当のところは、あなたが彼らを恐れる以上に、彼らはあなたを恐れている。ハッカーが集まってオフィスを借り、セールスマンを雇うのは、規模によらず会社がソフトウェアを書き上げることよりもずっと易しい。私はどちらの側も経験したから知っている。**Viaweb** が **Yahoo** に買収されて、私は突如として大企業で働くことになった。そこでの経験は、腰まで水に浸かった状態で走ろうとしているような感じだった。

I don't mean to disparage Yahoo. They had some good hackers, and the top management were real butt-kickers. For a big company, they were exceptional. But they were still only about a tenth as productive as a small startup. No big company can do much better than that. What's scary about Microsoft is that a company so big can develop software at all. They're like a mountain that can walk. いや、**Yahoo** をけなすつもりはないよ。素晴らしいハッカーたちがいたし、マネジメントのトップはすごい連中だった。大企業としては、彼らは例外的な部類に入るだろう。それ

でも、彼らは生産性は小回りのきくベンチャー企業のほんの 1/10 程度しかなかったと思う。大企業では高い生産性は望めない。マイクロソフトがすごいのは、あれだけの図体にもかかわらず、とにもかくにもソフトウェアを開発できているという事実だ。まるで、歩くことができる山のような。

Don't be intimidated. You can do as much that Microsoft can't as they can do that you can't. And no one can stop you. You don't have to ask anyone's permission to develop Web-based applications. You don't have to do licensing deals, or get shelf space in retail stores, or grovel to have your application bundled with the OS. You can deliver software right to the browser, and no one can get between you and potential users without preventing them from browsing the Web.

しかし、びくびくしないことだ。あなたはマイクロソフトのすることはできないが、マイクロソフトだってあなたがすることができないのだから。そして、誰もあなたを止められない。Web ベースのアプリケーションを書くのに、誰の許可もいらない。ライセンスを取る手続きだとか、小売店の陳列棚を確保したりだとか、OS にアプリケーションをバンドルしてもらうように交渉するだとか、そういうことは一切必要ないんだ。ソフトウェアは直接ブラウザにダウンロードしてもらえばいい。その邪魔をするには、人々が Web をブラウズすることを禁止するしかないが、そんなことなどできっこない。



Bill Gates, 1977

You may not believe it, but I promise you, Microsoft is scared of you. The complacent middle managers may not be, but Bill is, because he was you once, back in 1975, the last time a new way of delivering software appeared.

信じられないかもしれないが、マイクロソフトは間違いなくあなたを恐れている。中間管理職の脳転気な連中はさておいても、ビル・ゲイツはきっとそうだ。今のあなたは、新しいソフトウェア開発法が生まれた 1975 年当時の彼と同じなのだから。

Notes

注釈

[1] Realizing that much of the money is in the services, companies building lightweight clients have usually tried to combine the hardware with an [online service](#). This approach has not worked well, partly because you need two different kinds of companies to build consumer electronics and to run an online service, and partly because users hate the idea. Giving away the razor and making money on the blades may work for Gillette, but a razor is much smaller commitment than a Web terminal. Cell phone handset makers are satisfied to sell hardware without trying to capture the service revenue as well. That should probably be the model for Internet clients too. If someone just sold a nice-looking little box with a Web browser that you could use to connect through any ISP, every technophobe in the country would buy one.

[1] 軽量クライアントを作る会社の多くは、多くの金がサービスに支払われるようになるということに気付くと、ハードウェアとオンラインサービスを組み合わせようとした。このアプローチはうまくない。家電機器を作るのとオンラインサービスを提供するのには全く違った種類の会社が必要だというのがひとつの理由で、ユーザがそういう抱き合わせを嫌がるというのがもうひとつの理由だ。かみそりをただで提供して替え刃で儲けるという戦略はジレット社にとってはうまくいったが、かみそりは Web 端末よりはるかに小さなコミットメントだ。携帯電話のハンドセットのメーカーは、サービスによる収入をも手に入れようとはせずに、ハードを売って商売になっている。インターネットクライアントもそういうモデルになるべきなんだろう。誰かが、Web ブラウザ内蔵でどんな ISP にもつなげるような格好良いデバイス売り出したら、国中の新しもの好きが買うに違いない。

[2] Security always depends more on not screwing up than any design decision, but the nature of server-based software will make developers pay more attention

to not screwing up. Compromising a server could cause such damage that ASPs (that want to stay in business) are likely to be careful about security.

[2] セキュリティは、他のどんなデザイン上の決断にもまして、「失敗しないこと」が重要だ。サーバベースのソフトウェアでは、原理的に開発者は失敗しないように自然に注意を払うようになる。サーバに侵入を許すことは、ASP にあまりに巨大な被害を与えるため(もちろん彼らはビジネスを続けたいだろうから)、とりわけセキュリティには十分に注意するようになるだろう。

[3] In 1995, when we started Viaweb, Java applets were supposed to be the technology everyone was going to use to develop server-based applications. Applets seemed to us an old-fashioned idea. Download programs to run on the client? Simpler just to go all the way and run the programs on the server. We wasted little time on applets, but countless other startups must have been lured into this tar pit. Few can have escaped alive, or Microsoft could not have gotten away with dropping Java in the most recent version of Explorer.

[3] 私たちが Viaweb を立ち上げた 1995 年当時、Java アプレットがサーバベースのアプリケーションを開発するのに誰もが使う技術だと思われていた。でも、私たちにとっては、アプレットは古いアイディアのように思えた。クライアントで走らせるためにプログラムをダウンロードするだって? すっぱりと全てサーバ側でプログラムを走らせるようにした方が簡単じゃないか。というわけで私たちはアプレットで時間を潰すようなことはしなかった。でも、数え切れないほどのベンチャーがこの底無し沼に誘われたに違いない。そこから生きて抜け出せたものはほとんどなかった。でなければ、マイクロソフトが最新の Explorer で Java のサポートを削るなんてことは不可能だったろう。

[4] This point is due to Trevor Blackwell, who adds "the cost of writing software goes up more than linearly with its size. Perhaps this is mainly due to fixing old bugs, and the cost can be more linear if all bugs are found quickly."

[4] この点はトレバー・ブラックウェルが指摘した。彼はさらにこうも言っている。「ソフトウェアを書くコストは、サイズが増加するに従って、より急速な勢いで上昇する。たぶんその主要な要因は古いバグを修正しなければならないためだ。バグが全て素早く見つかるなら、コストは線形に近付くだろう。」

[5] The hardest kind of bug to find may be a variant of compound bug where one bug happens to compensate for another. When you fix one bug, the other becomes visible. But it will seem as if the fix is at fault, since that was the last thing you changed.

[5] 最も厄介なバグは、複合バグの一種で、ひとつのバグが別のバグの不具合をたまたま隠してしまうようなものだ。一方を直すと、もう一方が現れる。でも、それはまるで最初の修正が間違っていたかのように思いがちだ。それが最後に変更したところだから。

[6] Within Viaweb we once had a contest to describe the worst thing about our software. Two customer support people tied for first prize with entries I still shiver to recall. We fixed both problems immediately.

[6] Viaweb 内で、自分たちのソフトウェアについて最悪のことを挙げる、というコンテストを行ったことがある。カスタマーサポートのスタッフ 2 人が、同票で 1 位を分け合った。彼らの挙げた内容は、今思い出しても震えがくるほどだ。それをすぐに直したのは言うまでもない。

[7] Robert Morris wrote the ordering system, which shoppers used to place orders. Trevor Blackwell wrote the image generator and the manager, which merchants used to retrieve orders, view statistics, and configure domain names etc. I wrote the editor, which merchants used to build their sites. The ordering system and image generator were written in C and C++, the manager mostly in Perl, and the editor in [Lisp](#).

[7] ロバート・モリスは、買い物客が注文を出すシステムを書いた。トレバー・ブラックウェルは店側が注文を取り出したり、統計情報を見たり、ドメイン名を設定したりするための画像生成部と管理プログラムを書いた。私は店が自分のオンラインショップを構築するためのエディタを書いた。注文システムと画像生成部は C と C++ で、管理プログラムは主として Perl で、そしてエディタは Lisp で書かれていた。

[8] Price discrimination is so pervasive (how often have you heard a retailer claim that their buying power meant lower prices for you?) that I was surprised to find it was outlawed in the U.S. by the Robinson-Patman Act of 1936. This law does not appear to be vigorously enforced.

[8] 価格差別は広く浸透している(小売店が、彼らの購買力によって消費者により安い価格を提供できます、と宣伝しているのを何度も聞いたことがあるだろう)。だから、実はそれが米国では 1936 年の Robinson-Patman 法により禁止されていることを知って私はびっくりした。ただ、この法律はあまり厳格に運用されていないようだ。

[9] In *No Logo*, Naomi Klein says that clothing brands favored by "urban youth" do not try too hard to prevent shoplifting because in their target market the shoplifters are also the fashion leaders.

[9] ナオミ・クラインの『ブランドなんか、いらない』では、都市部の若者に好まれるブランドの衣料店は万引きを防ぐ工夫をそれほどきつくしないということが述べられている。それらブランドのマーケットにおいては、万引き犯は往々にしてファッションリーダーでもあるからだ。

[10] Companies often wonder what to outsource and what not to. One possible answer: outsource any job that's not directly exposed to competitive pressure, because outsourcing it will thereby expose it to competitive pressure.

[10] 何をアウトソースすべきで、何をすべきでないかは、会社にとってしばしば問題となる。こういう解答はどうだろう。競争的なプレッシャーに晒されていない職種を全てアウトソースする。アウトソースすることで、そういう職種も競争的なプレッシャーに晒すことができる。

[11] The two guys were Dan Bricklin and Bob Frankston. Dan wrote a prototype in Basic in a couple days, then over the course of the next year they worked together (mostly at night) to make a more powerful version written in 6502 machine language. Dan was at Harvard Business School at the time and Bob nominally had a day job writing software. "There was no great risk in doing a business," Bob wrote, "If it failed it failed. No big deal."

[11] その2人とは、ダン・ブリックリンとボブ・フランクストンだ。ダンがプロトタイプを Basic を使って2日ばかりで書き上げ、それから翌年にかけて、2人は(主に夜間に)より強力なバージョンを 6502 の機械語で書き上げた。ダンが当時ハーバードビジネススクールの学生で、ボブはいわゆる昼の仕事としてもソフトウェアを書いていた。「ビジネスをやることにそんなに大きなリスクはなかった。」とボブは書いている。「失敗したら、それまでさ。どうってことない。」

[12] It's not quite as easy as I make it sound. It took a painfully long time for word of mouth to get going, and we did not start to get a lot of press coverage until we hired a [PR firm](#) (admittedly the best in the business) for \$16,000 per month. However, it was true that the only significant channel was our own Web site.

[12] もっとも、ここでさらっと言う程簡単なことではなかった。口コミがひろがるのにはひどく時間がかかったし、マスコミに取り上げられたのは、私たちがようやく広告会社(その分野では最高の会社だったが)を、月に\$16,000という費用で雇ってからだ。ただ、私たちの主要な顧客獲得窓口が私たちの Web サイトであった、ということは間違いない。

[13] If the Mac was so great, why did it lose? Cost, again. Microsoft concentrated

on the software business, and unleashed a swarm of cheap component suppliers on Apple hardware. It did not help, either, that suits took over during a critical period.

[13] Mac がそんなに良いものなら、どうして負けたんだろう?コストだ。またしても。マイクロソフトはソフトウェアビジネスに集中し、アップルのハードウェアに対しては、大量の安い部品業者に対抗させたんだ。もちろん、重要な時期にスーツ族が会社の舵取りをしていたというのも原因のひとつではあるが。

[14] One thing that would help Web-based applications, and help keep the next generation of software from being overshadowed by Microsoft, would be a good open-source browser. Mozilla is open-source but seems to have suffered from having been corporate software for so long. A small, fast browser that was actively maintained would be a great thing in itself, and would probably also encourage companies to build little Web appliances.

[14] Web ベースアプリケーションの追い風になり、また次世代のソフトウェアをマイクロソフトの影から開放するのに役立つものの一つは、オープンソースの良いブラウザだろう。Mozilla はオープンソースではあるが、長い間、会社で作られるソフトウェアであったことからの影響をまだ抜け切れないようだ。小さく、軽いブラウザがあれば、それ自体が良いものであるだけでなく、ちょっとした Web アプライアンスを作ろうという会社を助けることにもなるだろう。

Among other things, a proper open-source browser would cause HTTP and HTML to continue to evolve (as e.g. Perl has). It would help Web-based applications greatly to be able to distinguish between selecting a link and following it; all you'd need to do this would be a trivial enhancement of HTTP, to allow multiple urls in a request. Cascading menus would also be good.

さらに、正しいオープンソースのブラウザは HTTP と HTML が進化し続けることを助けるだろう(例えば Perl がそうしたように)。Web ベースのアプリケーションが、例えばリンクを選択することとそれをたどることを区別できるようになることで、Web ベースのアプライアンスは大いに助かるだろう。必要なのは、リクエストに複数の URL を指定できるようにする、HTTP のちょっとした拡張だけだ。複数段のメニューもあったら良いだろう。

If you want to change the world, write a new Mosaic. Think it's too late? In 1998 a lot of people thought it was too late to launch a new search engine, but Google proved them wrong. There is always room for something new if the current options suck enough. Make sure it works on all the free OSes first-- new things

start with their users.

世界を変えたければ、新しい Mosaic を書くことだ。もう遅すぎるって?1998 年には、多くの人は新しいサーチエンジンを出すのは遅すぎると思っていたが、Google はそれが間違っていることを証明した。現在のものより格段に素晴らしければ、新しいものはいつでも歓迎される。もしそれを作るなら、まず全てのフリーな OS での動作を保証することだ。新しいことは、往々にしてフリーな OS のユーザから始まる。

[15] Trevor Blackwell, who probably knows more about this from personal experience than anyone, writes:

[15] おそらく、個人的な経験を通じて誰よりもこの問題を知っているであろうトレバー・ブラックウェルはこう書いている。

"I would go farther in saying that because server-based software is so hard on the programmers, it causes a fundamental economic shift away from large companies. It requires the kind of intensity and dedication from programmers that they will only be willing to provide when it's their own company. Software companies can hire skilled people to work in a not-too-demanding environment, and can hire unskilled people to endure hardships, but they can't hire highly skilled people to bust their asses. Since capital is no longer needed, big companies have little to bring to the table."

「僕なら、もう一歩進めて、サーバベースのソフトウェアはプログラマにとって非常に難しく、したがって大企業が離れて行くという根本的な経済上の変革をもたらすと言うだろうね。サーバベースソフトウェアは、プログラマにもものすごい密度で全身全霊をかけることを要求するから、自分の会社だというのでもなければとてもやる気にならないさ。スキルの高いプログラマを雇って、それほどきつくない環境で働いてもらうことはできる。また、スキルのないプログラマを雇って馬車馬のように働かせることもできる。でも、非常に高いスキルを持つプログラマを雇って馬車馬のように働かせることなんてできやしない。資本が必要ないこのフィールドには、大会社が持ち込めるものなんてほとんどないさ。」

[16] In the original version of this essay, I advised avoiding Javascript. That was a good plan in 2001, but Javascript now works.

[16] このエッセイの最初の版では、私は JavaScript を避けることを推奨している。それは 2001 年の時点では良いことだった。しかし、現在では JavaScript は使える。

Thanks to Sarah Harlin, Trevor Blackwell, Robert Morris, Eric Raymond, Ken Anderson, and Dan Giffin for reading drafts of this paper; to Dan Bricklin and Bob

Frankston for information about VisiCalc; and again to Ken Anderson for inviting me to speak at BBN.

Chapter 6 HOW TO MAKE WEALTH

第6章 富の創りかた

May 2004

(This essay was originally published in [Hackers & Painters](#).)

(このエッセイはもともと「ハッカーと画家」に収録されていたものである。)

If you wanted to get rich, how would you do it? I think your best bet would be to start or join a startup. That's been a reliable way to get rich for hundreds of years. The word "startup" dates from the 1960s, but what happens in one is very similar to the venture-backed trading voyages of the Middle Ages.

裕福になりたいと思ったらどうすればいい?一番分の良い方法はおそらく、ベンチャー企業(スタートアップ)を起こすか、それに参加するかだ。これはここ数百年にわたって、裕福になる信頼性の高い方法である。「スタートアップ」という言葉が使われるようになったのは1960年代からだが、その中身は、中世に資本家がスポンサーとなっていた貿易航海で行われていたこととほとんど同じだ。

Startups usually involve technology, so much so that the phrase "high-tech startup" is almost redundant. A startup is a small company that takes on a hard technical problem.

ベンチャー企業は通常、新しい技術と関連がある。だから「ハイテクベンチャー」というのは冗長な言い回しとも言える。ベンチャー企業とは、困難な技術的問題に挑戦する小さな企業だ。

Lots of people get rich knowing nothing more than that. You don't have to know physics to be a good pitcher. But I think it could give you an edge to understand the underlying principles. Why do startups have to be small? Will a startup inevitably stop being a startup as it grows larger? And why do they so often work on developing new technology? Why are there so many startups selling new drugs or computer software, and none selling corn oil or laundry detergent?

専門以外のことは何も知らないでも裕福になった人々はたくさんいる。良い投手になるのに物理学を知る必要はない。でも、基礎となる原理を理解しておいたほうが、有利な立場に立てるんじゃないかと私は思う。なぜベンチャー企業は小さくなくてはならないんだろう。なぜ企業は大きくなると必然的にベンチャー企業でなくなるのだろう。そして、なぜそういう新興企業は新しい技術の開発に取り組むんだろう。ベンチャー企業のほとんどが、

サラダ油や洗剤ではなく、新薬やコンピュータソフトウェアを売っているのはなぜなんだろう。

The Proposition

命題

Economically, you can think of a startup as a way to compress your whole working life into a few years. Instead of working at a low intensity for forty years, you work as hard as you possibly can for four. This pays especially well in technology, where you earn a premium for working fast.

経済的には、ベンチャー企業は一生分の労働時間を数年間に圧縮したものだと考えられる。ゆったりと 40 年働く代わりに、可能な限りハードに 4 年間働くんだけ。技術の分野では速い仕事により価値があるので、この手法はとりわけ有利だ。

Here is a brief sketch of the economic proposition. If you're a good hacker in your mid twenties, you can get a job paying about \$80,000 per year. So on average such a hacker must be able to do at least \$80,000 worth of work per year for the company just to break even. You could probably work twice as many hours as a corporate employee, and if you focus you can probably get three times as much done in an hour. [1] You should get another multiple of two, at least, by eliminating the drag of the pointy-haired middle manager who would be your boss in a big company. Then there is one more multiple: how much smarter are you than your job description expects you to be? Suppose another multiple of three. Combine all these multipliers, and I'm claiming you could be 36 times more productive than you're expected to be in a random corporate job. [2] If a fairly good hacker is worth \$80,000 a year at a big company, then a smart hacker working very hard without any corporate bullshit to slow him down should be able to do work worth about \$3 million a year.

この経済的な命題を簡単に述べてみよう。あなたが 20 代半ばの優秀なハッカーだとすると、おそらく年棒 8 万ドルくらいの仕事を見つけられるだろう。ということは、そのハッカーが平均して会社に 8 万ドルの利益をもたらせば、会社としてはトントンということになる。さて、やろうと思えばおそらくあなたは、従業員として働く時間の 2 倍の時間を仕事に充てることができる。そして集中すれば、おそらく 1 時間当たりの生産性は 3 倍くらいに上げられるだろう。[1] また、大企業にいる、髪のとんがった中間管理職の上司に足を引っ張られなければ、少なく見積もってさらに 2 倍の仕事ができる。さらにもうひとつ係数がある。求人票に出ていた仕事内容に比べて、あなたはどのくらい優秀だろうか。この係数

を 3 倍と見積もろう。全てを合わせると、適当な会社に入って仕事をするのに比べて、あなたは 36 倍の仕事をする能力があるということになる。[\[2\]](#) もし、かなり優秀なハッカーが大企業で 8 万ドルの年棒に値するとしたら、その人物が、足手まといになる企業内のごたごたに縛られずに仕事をすれば年間 300 万ドルの仕事ができることになる。

Like all back-of-the-envelope calculations, this one has a lot of wiggle room. I wouldn't try to defend the actual numbers. But I stand by the structure of the calculation. I'm not claiming the multiplier is precisely 36, but it is certainly more than 10, and probably rarely as high as 100.

もちろん、実際には、これ以外にも色々な要因がからんでくるはずなので、この数値そのものの正当性を主張するつもりはない。でも計算式自体は間違っていないと思う。係数が 36 ぴったりであるなんて言うつもりはないが、たぶんそれは 10 より大きいだろうし、かといって 100 まではいかないだろう。

If \$3 million a year seems high, remember that we're talking about the limit case: the case where you not only have zero leisure time but indeed work so hard that you endanger your health.

年間 300 万ドルというのが大き過ぎる数字に思えるかもしれないが、これはあくまでも限界まで働いた場合、つまり、自分のための余暇の時間は全くなく、そして健康を損ねかねないほどハードに働いた場合の数字、ってことだ。

Startups are not magic. They don't change the laws of wealth creation. They just represent a point at the far end of the curve. There is a conservation law at work here: if you want to make a million dollars, you have to endure a million dollars' worth of pain. For example, one way to make a million dollars would be to work for the Post Office your whole life, and save every penny of your salary. Imagine the stress of working for the Post Office for fifty years. In a startup you compress all this stress into three or four years. You do tend to get a certain bulk discount if you buy the economy-size pain, but you can't evade the fundamental conservation law. If starting a startup were easy, everyone would do it.

ベンチャー企業は魔法じゃない。富の生成の法則を変えることはできない。曲線の一番端の点を表現しているにすぎないんだ。ここにも保存則が適用される。100 万ドル儲けたければ、100 万ドル分の苦勞をしなくちゃならない。例えば郵便局に一生勤めて、給料の最後の 1 セントまでをこつこつ貯金すれば、100 万ドルを貯めることはできるだろう。そうやって 50 年間、郵便局に勤めるストレスを想像してほしい。ベンチャー企業では、同じ量のストレスが 3~4 年間に圧縮されてやってくるんだ。まとめ買いによる割引が利く場合も

あるが、根本的な保存則を変えることはできない。ベンチャー企業を立ち上げることが簡単なら、みんなやってるはずだろう？

Millions, not Billions

億万長者ではなく、百万長者

If \$3 million a year seems high to some people, it will seem low to others. Three *million*? How do I get to be a billionaire, like Bill Gates?

年棒 300 万ドルは、人によってはとても多く見えるだろうが、そうでもないと思う人もいるだろう。たった 300 万だって？それより、どうやったらビル・ゲイツみたいに、何十億もの財産を作ることができるんだ？

So let's get Bill Gates out of the way right now. It's not a good idea to use famous rich people as examples, because the press only write about the very richest, and these tend to be outliers. Bill Gates is a smart, determined, and hardworking man, but you need more than that to make as much money as he has. You also need to be very lucky.

とりあえずビル・ゲイツは置いておこう。有名で裕福な人物を例として使うのは良くない。というのも、メディアはなかでも一番裕福な人々についてしか書かないし、そういう人々は例外点であることが多いからだ。確かに、ビル・ゲイツは明晰な頭脳と強固な意志を持ち、ハードに働く人間だが、それだけで彼ほどの財を築けるわけではない。恐ろしいほどの幸運というものも必要なんだ。

There is a large random factor in the success of any company. So the guys you end up reading about in the papers are the ones who are very smart, totally dedicated, *and* win the lottery. Certainly Bill is smart and dedicated, but Microsoft also happens to have been the beneficiary of one of the most spectacular blunders in the history of business: the licensing deal for DOS. No doubt Bill did everything he could to steer IBM into making that blunder, and he has done an excellent job of exploiting it, but if there had been one person with a brain on IBM's side, Microsoft's future would have been very different. Microsoft at that stage had little leverage over IBM. They were effectively a component supplier. If IBM had required an exclusive license, as they should have, Microsoft would still have signed the deal. It would still have meant a lot of money for them, and IBM could easily have gotten an operating system elsewhere.

どんな会社の成功にも、大きなランダム要素が絡んでくる。だからメディアで報道される

ような成功者は、明晰な頭脳と強固な意志、それに宝くじに当たるような幸運を併せ持った人なんだ。マイクロソフトが大きな利益を上げることができたのは、ビル・ゲイツが明晰な頭脳と強固な意志を持ち、取引相手がビジネス史上稀に見る大失策を犯したからだ。DOS のライセンス契約だ。もちろんビル・ゲイツは総力をあげて IBM がその失策をするように仕向けただろうし、それを利用するに当たっても素晴らしい仕事をしたが、もし IBM 側に一人でも頭脳を持った人間がいたなら、マイクロソフトの将来は全く違ったものになっていただろう。マイクロソフトはあの時点では IBM に対してさしたる影響力を持っていなかった。実質的に、部品の供給業者にすぎなかったんだ。もし IBM がそうすべきだったように独占契約を要求していたとしても、マイクロソフトはそれを飲んでいただろう。そうだったとしても当時の彼らにとっては大金だったし、IBM はその気になれば他からオペレーティングシステムを探してくることだってできたのだから。

Instead IBM ended up using all its power in the market to give Microsoft control of the PC standard. From that point, all Microsoft had to do was execute. They never had to bet the company on a bold decision. All they had to do was play hardball with licensees and copy more innovative products reasonably promptly.

だが IBM は結局、市場に持てる力を全て使って、マイクロソフトに PC の標準を握らせてしまった。マイクロソフトは、そこからはただ実行するだけでよかった。企業を賭けた難しい決断を迫られることはなかったんだ。ライセンス先の企業から取れるだけ取り、最新の技術を十分に早くコピーしてさえいればよかった。

If IBM hadn't made this mistake, Microsoft would still have been a successful company, but it could not have grown so big so fast. Bill Gates would be rich, but he'd be somewhere near the bottom of the Forbes 400 with the other guys his age.

IBM がこの過ちを犯していなくても、マイクロソフトはまあまあ成功した企業になっていただろうが、こんなに早く大きくはなっていなかっただろう。ビル・ゲイツは裕福になっていただろうけれど、たぶん Forbes 400 の下のほうに、同年代の起業家と一緒に並んでいたんじゃないかと思う。

There are a lot of ways to get rich, and this essay is about only one of them. This essay is about how to make money by creating wealth and getting paid for it. There are plenty of other ways to get money, including chance, speculation, marriage, inheritance, theft, extortion, fraud, monopoly, graft, lobbying, counterfeiting, and prospecting. Most of the greatest fortunes have probably involved several of these.

裕福になるには実にいろいろな方法があるが、このエッセイで扱うのはそのうちのひとつだけだ。このエッセイでは、富を創り、その対価を得ることで金を儲けることについて扱う。金を手に入れる方法は他にいくらでもある。偶然、投機、結婚、相続、窃盗、強盗、詐欺、独占、汚職、癒着、賄賂、炭鉱、等々。最大に財産のいくつかは、おそらくこれらのうちの数種類にかかわっているんじゃないかと思う。

The advantage of creating wealth, as a way to get rich, is not just that it's more legitimate (many of the other methods are now illegal) but that it's more *straightforward*. You just have to do something people want.

このうちで、富を生むことで裕福になる方法は、単により合法的というだけじゃない(他の方法の多くは現代では非合法だ)。より直接的なんだ。人が望むことをするだけでいいんだ。

Money Is Not Wealth

お金は富ではない

If you want to create wealth, it will help to understand what it is. Wealth is not the same thing as money. [\[3\]](#) Wealth is as old as human history. Far older, in fact; ants have wealth. Money is a comparatively recent invention.

富を築き上げようと思ったら、まずそれが何かを理解することが役に立つ。富はお金と同義ではない。[\[3\]](#) 富は、人類の歴史と同じくらい古くからある、いや、それより古くからあると言ってもよいだろう。蟻だって富を持っている。お金(貨幣)は比較的最近発明されたものだ。

Wealth is the fundamental thing. Wealth is stuff we want: food, clothes, houses, cars, gadgets, travel to interesting places, and so on. You can have wealth without having money. If you had a magic machine that could on command make you a car or cook you dinner or do your laundry, or do anything else you wanted, you wouldn't need money. Whereas if you were in the middle of Antarctica, where there is nothing to buy, it wouldn't matter how much money you had.

富はもっと根本的なものだ。富とは、私たちが欲しがらるもの、食物、衣服、住居、車、道具、観光地への旅行、そういったものだ。お金を持っていなくても富を持つことはできる。例えば、車を作れだとか夕食を料理してくれだとか洗濯をしてくれだとか、何でも命令すればやってくれる魔法の機械があったとしたら、もうお金はいらないだろう。一方、南極大陸のど真ん中、何も買い物ができない場所にいるとしたら、いくらお金を持っていたとしても無意味だ。

Wealth is what you want, not money. But if wealth is the important thing, why does everyone talk about making money? It is a kind of shorthand: money is a way of moving wealth, and in practice they are usually interchangeable. But they are not the same thing, and unless you plan to get rich by counterfeiting, talking about *making money* can make it harder to understand how to make money.

富こそが欲しいものなのであって、お金ではない。でも富がそんなに重要なら、どうしてみんな金を儲けることについて話しているんだろう。それは一種の省略記法なんだ。お金は富を移動する方法のひとつで、現実では普通、富と交換可能だ。しかし両者は違うものだ。「お金を作る」ことについて話すのは、ニセ札作りでもしようというのでなくや、むしろ実際に裕福になる方法を理解する妨げになる。

Money is a side effect of specialization. In a specialized society, most of the things you need, you can't make for yourself. If you want a potato or a pencil or a place to live, you have to get it from someone else.

お金は専門化の副作用だ。専門化された社会では、人が必要とする多くのものを自分で作り出すことはできない。じゃがいもや、鉛筆や、住む所が欲しければ、普通はそれを誰かからもらわなければならない。

How do you get the person who grows the potatoes to give you some? By giving him something he wants in return. But you can't get very far by trading things directly with the people who need them. If you make violins, and none of the local farmers wants one, how will you eat?

じゃがいもを作っている人から、いくらか分けてもらうにはどうすればいいだろう。その人が欲しがるものを代わりにあげればいい。でも互いに欲しがるものを直接交換しているだけでは、まだ不便なことこの上ない。例えばあなたがバイオリンを作っていて、近所の農家の人は誰もそれを欲しがらなかったとしたら、どうやって食べていける？

The solution societies find, as they get more specialized, is to make the trade into a two-step process. Instead of trading violins directly for potatoes, you trade violins for, say, silver, which you can then trade again for anything else you need. The intermediate stuff-- the *medium of exchange*-- can be anything that's rare and portable. Historically metals have been the most common, but recently we've been using a medium of exchange, called the *dollar*, that doesn't physically exist. It works as a medium of exchange, however, because its rarity is guaranteed by the U.S. Government.

社会が専門化してゆく過程で発明された解法は、交換を 2 段階にするというものだった。

バイオリンを直接じゃがいもと交換する代わりに、それを例えば銀と交換する。そしてその銀を必要な他のものとまた交換する。この中間に使われるもの、交換媒体は、希少で持ち運びができるものなら何でもよい。歴史的には金属が広く使われてきたが、近年、私たちは実際に物理的には存在しない「ドル」と呼ばれる交換媒体を使うようになった。それは物理的に存在しないが交換媒体として利用できる。米国政府がその希少性を保証しているからだ。

The advantage of a medium of exchange is that it makes trade work. The disadvantage is that it tends to obscure what trade really means. People think that what a business does is make money. But money is just the intermediate stage-- just a shorthand-- for whatever people want. What most businesses really do is make wealth. They do something people want. [\[4\]](#)

交換媒体の利点は、それにより交易が可能になるというものだ。しかし交易の本当の意味が分かりにくくなるという欠点もある。人々はビジネスというのはお金を儲けることだと思い始めた。だが、お金は欲しいものを手に入れるための単なる中間段階、省略記法にすぎない。ほとんどのビジネスがやっていることは富を生み出すことだ。人々が欲しがることをやるんだ。 [\[4\]](#)

The Pie Fallacy

パイの誤り

A surprising number of people retain from childhood the idea that there is a fixed amount of wealth in the world. There is, in any normal family, a fixed amount of *money* at any moment. But that's not the same thing.

驚くほど多くの人々が、子供のころ持った、世界には限られた富しかないという考えを持ち続けている。普通の家庭のある時点では、確かに限られたお金しかない。でもそれは同じことじゃない。

When wealth is talked about in this context, it is often described as a pie. "You can't make the pie larger," say politicians. When you're talking about the amount of money in one family's bank account, or the amount available to a government from one year's tax revenue, this is true. If one person gets more, someone else has to get less.

そういう文脈で富が語られるとき、それはよくパイに例えられる。政治家はよく、「パイを大きくすることはできない」と言う。ある家族の銀行口座にあるお金とか、政府の年間の税収とかについての話ならば、それは真実だ。誰かが余計に取ればその分誰かが損をする

ことになる。

I can remember believing, as a child, that if a few rich people had all the money, it left less for everyone else. Many people seem to continue to believe something like this well into adulthood. This fallacy is usually there in the background when you hear someone talking about how x percent of the population have y percent of the wealth. If you plan to start a startup, then whether you realize it or not, you're planning to disprove the Pie Fallacy.

私も子供のころ、もし少数の裕福な人々が全部のお金を取っちゃったら、他の皆には少ししか残されないじゃないかと思ったのを覚えている。どうも大人になってもこのことを信じている人が多いようだ。人口の x パーセントが y パーセントの富を所有している、といった議論の背景には、たいていこの誤りがある。もしあなたがベンチャー企業を立ち上げようとしているなら、自覚の有無にかかわらず、このパイの誤りを証明しようとしていることになる。

What leads people astray here is the abstraction of money. Money is not wealth. It's just something we use to move wealth around. So although there may be, in certain specific moments (like your family, this month) a fixed amount of money available to trade with other people for things you want, there is not a fixed amount of wealth in the world. *You can make more wealth.* Wealth has been getting created and destroyed (but on balance, created) for all of human history. 人々が間違える理由は、お金による抽象化だ。お金は富ではない。富を移動する手段にすぎないんだ。だから、ある時点(例えば今月の家計)に決まった量のお金しか他のものと交換するのに使えないからといって、世界中に決まった量の富しかないということにはならない。富は創り出すことができるんだ。人類の歴史のどの時点でも、富は生み出されたり、消滅したりして(だが、トータルとしては創り出されて)きた。

Suppose you own a beat-up old car. Instead of sitting on your butt next summer, you could spend the time restoring your car to pristine condition. In doing so you create wealth. The world is-- and you specifically are-- one pristine old car the richer. And not just in some metaphorical way. If you sell your car, you'll get more for it.

例えばあなたがポンコツの古い車を持っているとしよう。次の夏休みにぼさっとしている代わりに、その車を修理して元の状態に戻したとしよう。そうすることであなたは富を創り出しているんだ。世界は、そしてとりわけあなたは、ひとつのクラシックカーの分だけ裕福になったわけだ。これは比喩的な意味じゃない。あなたがその車を売ることを考えれ

ば、より多くの金を受け取ることができるだろう。

In restoring your old car you have made yourself richer. You haven't made anyone else poorer. So there is obviously not a fixed pie. And in fact, when you look at it this way, you wonder why anyone would think there was. [5]

古い車を修理することであなたは自分を裕福にできた。でも他の人に損をさせてはいない。だから、決まった大きさのパイなんか明らかに存在しないんだ。実際、一度こういうふうを考えてみれば、いったいどうして他の人が決まったパイなんて考えるのか、不思議にならないかい。 [5]

Kids know, without knowing they know, that they can create wealth. If you need to give someone a present and don't have any money, you make one. But kids are so bad at making things that they consider home-made presents to be a distinct, inferior, sort of thing to store-bought ones-- a mere expression of the proverbial thought that counts. And indeed, the lumpy ashtrays we made for our parents did not have much of a resale market.

子供は、自覚こそしていないが、自分が富を創り出せることを知っている。誰かにプレゼントをあげたくて、でもお金がないとき、自分で何かを作るだろう。ただ子供は何かを作るのがあまり上手ではないから、自家製のプレゼントは店で売っているものとは違う、何か劣ったもののように思ってしまうんだ。そこに込められた気持ちが大事なんだとか何とかね。確かに、両親のために作った不格好な灰皿は、店で売ったって大した金にならないだろう。

Craftsmen

職人

The people most likely to grasp that wealth can be created are the ones who are good at making things, the craftsmen. Their hand-made objects become store-bought ones. But with the rise of industrialization there are fewer and fewer craftsmen. One of the biggest remaining groups is computer programmers.

富が創り出されるものだということに気が付きやすいのは、ものを作るのが上手な人々、職人だ。職人の手作りの作品は、店で売られるものになる。だが、工業化が進むにつれ、ものを作る職人はどんどん少なくなってきた。現存するなかで最も大きいグループのひとつが、コンピュータプログラマだ。

A programmer can sit down in front of a computer and *create wealth*. A good piece

of software is, in itself, a valuable thing. There is no manufacturing to confuse the issue. Those characters you type are a complete, finished product. If someone sat down and wrote a web browser that didn't suck (a fine idea, by the way), the world would be that much richer. [5b]

プログラマはコンピュータの前に座って富を創り出すことができる。良いソフトウェアはそれだけで価値のあるものだ。問題を複雑にする製造工程なんてものはない。プログラマが打ち込む一文字一文字が完全な製品を構成するんだ。例えばどこかのプログラマが軽快な Web ブラウザを書いたとしたら(実際これはいけるアイデアだと思う)、世界はその分だけ裕福になる。[5b]

Everyone in a company works together to create wealth, in the sense of making more things people want. Many of the employees (e.g. the people in the mailroom or the personnel department) work at one remove from the actual making of stuff. Not the programmers. They literally think the product, one line at a time. And so it's clearer to programmers that wealth is something that's made, rather than being distributed, like slices of a pie, by some imaginary Daddy.

会社の中では、全員が協力して人々が欲しがるものを作るという意味で、富を創り出している。しかし従業員の多くは(例えば郵便物の仕分けとか人事部などは)実際にものを作る場所からは 1 ステップ離れている。プログラマはそうではない。プログラマは文字通り、製品を頭の中から一行一行紡ぎ出すんだ。したがってプログラマにとっては、富は仮想的なお父さんが切り分けてくれるパイなんかではなく、創り出されるものだということはより明白だ。

It's also obvious to programmers that there are huge variations in the rate at which wealth is created. At Viaweb we had one programmer who was a sort of monster of productivity. I remember watching what he did one long day and estimating that he had added several hundred thousand dollars to the market value of the company. A great programmer, on a roll, could create a million dollars worth of wealth in a couple weeks. A mediocre programmer over the same period will generate zero or even negative wealth (e.g. by introducing bugs).

また、プログラマにとっては、富が創られるペースには非常に大きな差があるということも明白だろう。Viaweb には生産性の鬼のようなプログラマがいた。ある長い一日に彼がした仕事を見ていて、彼は会社の市場価値を数十万ドル増やしたな、と思ったことを覚えている。凄腕のプログラマは半月で百万ドル相当の仕事をする事だってできるだろう。平凡なプログラマは同じ期間で、ゼロどころかマイナスの富を創ることだってある(バグを入れるからね)。

This is why so many of the best programmers are libertarians. In our world, you sink or swim, and there are no excuses. When those far removed from the creation of wealth-- undergraduates, reporters, politicians-- hear that the richest 5% of the people have half the total wealth, they tend to think *injustice!* An experienced programmer would be more likely to think *is that all?* The top 5% of programmers probably write 99% of the good software.

これが、良いプログラムの多くが自由主義者であることの理由だろう。プログラマの世界は泳ぐか沈むかで、言い訳は通用しない。富が創り出される現場からはるかに離れた所にいる人々、例えば学生や記者、政治家たちは、最も富める 5% の人々が全世界の富の半分を所有していると聞いて「不当だ!」と考える。経験豊かなプログラマはむしろ「それだけ?」と思うだろう。たぶん最も優秀な 5% のプログラマが、世の中の良いソフトウェアの 99% を書いているんじゃないだろうか。

Wealth can be created without being sold. Scientists, till recently at least, effectively donated the wealth they created. We are all richer for knowing about penicillin, because we're less likely to die from infections. Wealth is whatever people want, and not dying is certainly something we want. Hackers often donate their work by writing open source software that anyone can use for free. I am much the richer for the operating system FreeBSD, which I'm running on the computer I'm using now, and so is Yahoo, which runs it on all their servers.

創られた富は、必ずしも売られる必要はない。少なくとも近年まで、科学者は創り出した富を実質的に寄付してきた。ペニシリンを知ることによって私たちは豊かになった。感染から死ぬ危険が減ったからだ。富は人々が欲しがらるもので、死なないことは確かに私たちが欲することだ。ハッカーはしばしば、オープンソースソフトウェアとして誰もがただで使えるようにすることで、自分の仕事を寄付している。私は FreeBSD を使うことでとても豊かになった。今、これを書いているコンピュータでも走らせているし、Yahoo! でもサーバでそれを走らせている。

What a Job Is

仕事とは何か

In industrialized countries, people belong to one institution or another at least until their twenties. After all those years you get used to the idea of belonging to a group of people who all get up in the morning, go to some set of buildings, and do things that they do not, ordinarily, enjoy doing. Belonging to such a group

becomes part of your identity: name, age, role, institution. If you have to introduce yourself, or someone else describes you, it will be as something like, John Smith, age 10, a student at such and such elementary school, or John Smith, age 20, a student at such and such college.

産業化の進んだ国では、人々は 20 歳代になるまで何かの組織に所属して過ごす。それだけの年月を過ごせば、人は、何かのグループに所属して、一緒に朝から同じ建物に行き、普通は面白くもないことをやらねばならないものだ、という考えに慣れてしまう。組織に所属することが自分のアイデンティティにさえなる。氏名、年齢、役職と所属、というわけだ。自己紹介するときや、誰かに紹介されるときには、こんなふうになる。ジョン・スミス、10 歳、どこそこ小学校の生徒です。あるいは、ジョン・スミス、20 歳、どこそこ大学の学生です、とね。

When John Smith finishes school he is expected to get a job. And what getting a job seems to mean is joining another institution. Superficially it's a lot like college. You pick the companies you want to work for and apply to join them. If one likes you, you become a member of this new group. You get up in the morning and go to a new set of buildings, and do things that you do not, ordinarily, enjoy doing. There are a few differences: life is not as much fun, and you get paid, instead of paying, as you did in college. But the similarities feel greater than the differences. John Smith is now John Smith, 22, a software developer at such and such corporation.

このジョン・スミス氏が学校を卒業した。彼は就職することになる。すると、就職するということが、まるでもうひとつの組織に所属することのように思えてしまう。表面的にはそれは大学と似ている。自分が働きたい企業を選んでそこに応募する。気に入られればその新しいグループのメンバーになる。皆と一緒に朝から同じ建物に行って、普通は面白くもないことをやらされる。違いも少しはある。暮らしは学生の時ほど楽しくなくなるけど、授業料を払う代わりに給料を貰えるんだ。でも違いよりも類似点のほうがどうやら多いようだ。ジョン・スミス氏はいまや、ジョン・スミス、22 歳、どこそこ会社のプログラマーです、となったわけだ。

In fact John Smith's life has changed more than he realizes. Socially, a company looks much like college, but the deeper you go into the underlying reality, the more different it gets.

実はジョン・スミス氏の人生には、彼が気付いている以上に大きな変化が起きたんだ。社会的に、会社は大学と同じように見えるかもしれないが、より深い現実へと入ってゆけば違いのほうが目についてくる。

What a company does, and has to do if it wants to continue to exist, is earn money. And the way most companies make money is by creating wealth. Companies can be so specialized that this similarity is concealed, but it is not only manufacturing companies that create wealth. A big component of wealth is location. Remember that magic machine that could make you cars and cook you dinner and so on? It would not be so useful if it delivered your dinner to a random location in central Asia. If wealth means what people want, companies that move things also create wealth. Ditto for many other kinds of companies that don't make anything physical. Nearly all companies exist to do something people want.

会社がやっていること、そして存続してゆくためにやらねばならないことは、金を稼ぐことだ。そしてほとんどの会社が金を稼ぐ方法は、富を創り出すことによってだ。会社はそれぞれ非常に専門化しているので、この共通項は見え難くなっているけれど、製造業だけが富を創り出しているんじゃない。例えば富の大きな要素は場所だ。前に挙げた、車や夕食を作ってくれる魔法の機械の例を覚えているかい。あの機械が作った夕食を中央アジアのランダムな場所に届けるようになっていたら、あまり使い道がないだろう。富が人々の欲しがるものなら、ものを移動する会社だって富を創り出している。物理的なものを造っていない他のさまざまな会社だって同様だ。ほとんど全ての会社は、人々が欲することをやるために存在しているんだ。

And that's what you do, as well, when you go to work for a company. But here there is another layer that tends to obscure the underlying reality. In a company, the work you do is averaged together with a lot of other people's. You may not even be aware you're doing something people want. Your contribution may be indirect. But the company as a whole must be giving people something they want, or they won't make any money. And if they are paying you x dollars a year, then on average you must be contributing at least x dollars a year worth of work, or the company will be spending more than it makes, and will go out of business.

そしてあなたが会社で働いていれば、それはあなたがやることでもある。でも会社では、別の層が下にある現実を隠してしまいがちだ。会社では、あなたがやる仕事は他の多くの人がやることと平均化されてしまうんだ。他人が欲することをやっているんだ、ということに自分では気付かないかもしれない。あなたの貢献は間接的かもしれない。でも会社全体としては、誰かが欲するものを提供していない限り、利益を上げることはできない。会社があなたに年棒 x ドル払っているとすれば、あなたは少なくとも平均して年に x ドル以上の貢献をしているはずだ。でなければ会社は稼ぐよりも多くのお金を使うことになり、潰れてしまうからね。

Someone graduating from college thinks, and is told, that he needs to get a job, as if the important thing were becoming a member of an institution. A more direct way to put it would be: you need to start doing something people want. You don't need to join a company to do that. All a company is a group of people working together to do something people want. It's doing something people want that matters, not joining the group. [6]

大学を卒業しようとしている学生は、就職しなくちゃならないと周りから言われるし、自分でもそう考える。まるで組織のメンバーになることが重要であるかのように。でももっと間接的に言えば、人々が欲することを始めることが必要なんだ。そのために会社に所属する必要はない。会社とは、人々が集まって他の人々が欲することをやる集団にすぎない。人々が欲することをすることが重要なのであって、集団に属することは問題じゃない。[6]

For most people the best plan probably is to go to work for some existing company. But it is a good idea to understand what's happening when you do this. A job means doing something people want, averaged together with everyone else in that company.

たいていの人々にとっては、既にある会社に就職して働くことが最良のプランだろう。それでも、そうすることで本当は何をしているかを理解しておくことは重要だ。仕事とは、会社の他の人々と一緒に平均化されつつ、人々が欲することをなすことだ。

Working Harder

もっと頑張って働く

That averaging gets to be a problem. I think the single biggest problem afflicting large companies is the difficulty of assigning a value to each person's work. For the most part they punt. In a big company you get paid a fairly predictable salary for working fairly hard. You're expected not to be obviously incompetent or lazy, but you're not expected to devote your whole life to your work.

問題なのはこの平均化の部分だ。大企業における唯一にして最大の問題は、各人の仕事の価値を測ることの困難さだと思う。たいていの場合、彼らは賭けに出る。大企業では、従業員にそこそこ予測可能な給料を支払い、そこそこ頑張って働くことを期待するんだ。従業員は、明白に仕事ができなかったり怠けたりするのは困るが、全人生を仕事に賭けることも期待されていない。

It turns out, though, that there are economies of scale in how much of your life

you devote to your work. In the right kind of business, someone who really devoted himself to work could generate ten or even a hundred times as much wealth as an average employee. A programmer, for example, instead of chugging along maintaining and updating an existing piece of software, could write a whole new piece of software, and with it create a new source of revenue.

しかし生活のどれだけの部分を仕事に捧げられるかに関しては、規模の経済が成り立つ。ビジネスの種類によっては、本当に仕事に打ち込めば、平均的な社員の 10 倍どころか 100 倍もの富を生み出すことさえできる。例えばプログラマは、既存のソフトウェアを保守したり改良したりして時間を潰す代わりに、全く新しいソフトウェアを書いて新しい利益の源を作り出すことだってできる。

Companies are not set up to reward people who want to do this. You can't go to your boss and say, I'd like to start working ten times as hard, so will you please pay me ten times as much? For one thing, the official fiction is that you are already working as hard as you can. But a more serious problem is that the company has no way of measuring the value of your work.

会社はそういうことをしたい人のためにはできていない。上司のところに行って、明日から 10 倍働くので給料を 10 倍にしてくださいとは言えない。そもそも表向きあなたは既に可能な限り頑張っていることになっている。だがより重大な問題は、会社はあなたの仕事の価値を測る手段を持っていないということなんだ。

Salesmen are an exception. It's easy to measure how much revenue they generate, and they're usually paid a percentage of it. If a salesman wants to work harder, he can just start doing it, and he will automatically get paid proportionally more.

セールスマンは例外だ。彼らがどれだけの利益を産んでいるかは簡単に測れるし、したがってそれに比例した給料を受け取るようになっていることが多い。セールスマンがもっとたくさん働きたいと思えば、ただそうすればいいし、それに従った給料を自動的に貰えるだろう。

There is one other job besides sales where big companies can hire first-rate people: in the top management jobs. And for the same reason: their performance can be measured. The top managers are held responsible for the performance of the entire company. Because an ordinary employee's performance can't usually be measured, he is not expected to do more than put in a solid effort. Whereas top management, like salespeople, have to actually come up with the numbers. The

CEO of a company that tanks cannot plead that he put in a solid effort. If the company does badly, he's done badly.

営業部門以外にもうひとつ、大企業が一級の人々を雇える職種がある。最高経営責任者だ。そしてそれもまた同じ理由による。彼らの成績は測ることができるからだ。最高経営責任者は会社全体の利益に責任を負う。普通の従業員は仕事の成果を測られないから、「ちゃんと仕事をする」以上のことは求められない。しかし最高経営責任者はセールスマンと同様、数字を出さなければならない。傾いた会社の CEO は「ちゃんと頑張ったんです」と言っていて駄目だ。会社の利益が上がらなければ、彼は仕事をうまくやれなかったということだ。

A company that could pay all its employees so straightforwardly would be enormously successful. Many employees would work harder if they could get paid for it. More importantly, such a company would attract people who wanted to work especially hard. It would crush its competitors.

会社が全ての従業員に仕事量通りの給料を支払えれば、その会社はおそらく成功するだろう。仕事した分だけ給料が貰えるとなれば、多くの従業員は頑張って働くはずだ。さらに重要なのは、そういう会社が特に頑張って働きたい人を惹きつけることだ。競争相手は打ち負かされるだろう。

Unfortunately, companies can't pay everyone like salesmen. Salesmen work alone. Most employees' work is tangled together. Suppose a company makes some kind of consumer gadget. The engineers build a reliable gadget with all kinds of new features; the industrial designers design a beautiful case for it; and then the marketing people convince everyone that it's something they've got to have. How do you know how much of the gadget's sales are due to each group's efforts? Or, for that matter, how much is due to the creators of past gadgets that gave the company a reputation for quality? There's no way to untangle all their contributions. Even if you could read the minds of the consumers, you'd find these factors were all blurred together.

残念ながら、会社は全員にセールスマンのように支払うことはできない。セールスマンは一人で仕事をするが、多くの従業員は一緒になって仕事する。例えば一般向けの便利な小型機器を作っている会社を考えてみよう。エンジニアは、頑丈でいろいろ新しい機能がついた機器を製造する。工業デザイナーが、素敵な外観をデザインする。そしてマーケティング部門が、消費者に、これは必須のアイテムだと信じさせる。この製品の売上のうち、各グループの努力がどのくらいを占めるのかというのをどう測ればいいのか。さらに、その会社のそれまでの製品によって培われた評判も考え合わせると、過去の製品の作成者の貢献だってある。これらの要素を解きほぐす方法なんてない。たとえ消費者の思考が読

めたとしたって、いろいろな要素は曖昧に交じりあって判然としないだろう。

If you want to go faster, it's a problem to have your work tangled together with a large number of other people's. In a large group, your performance is not separately measurable-- and the rest of the group slows you down.

速く仕事をしたいと思ったら、多人数と一緒に何かをやるのは問題になる。大きな集団では個人の成果は分離して測ることはできない。そして、速く行こうとする人の足を他のメンバーが引っ張るのだ。

Measurement and Leverage

測定と槌子

To get rich you need to get yourself in a situation with two things, measurement and leverage. You need to be in a position where your performance can be measured, or there is no way to get paid more by doing more. And you have to have leverage, in the sense that the decisions you make have a big effect.

裕福になるためには、2つの環境を整えなければならない。測定と槌子だ。まず、自分の生産性が測れる地位に就かなければならない。でないと頑張っただけ支払ってもらえないからだ。また、槌子が必要だ。すなわち、あなたの決定が大きな効果を持つようにすることだ。

Measurement alone is not enough. An example of a job with measurement but not leverage is doing piecework in a sweatshop. Your performance is measured and you get paid accordingly, but you have no scope for decisions. The only decision you get to make is how fast you work, and that can probably only increase your earnings by a factor of two or three.

測定だけでは十分ではない。測定があつて槌子がない職種とは、例えば工場での歩合制の単純作業だ。仕事ぶりは測定されて、それに見合った給料が支払われるが、決断することはほとんど何もない。唯一決められることはどれだけの仕事を自分でするかだが、それだけでは頑張ってもせいぜい収入を2倍か3倍にするだけだろう。

An example of a job with both measurement and leverage would be lead actor in a movie. Your performance can be measured in the gross of the movie. And you have leverage in the sense that your performance can make or break it.

測定と槌子が両方ある仕事の例は、映画俳優だ。仕事ぶりは映画の興行収入で測られる。そして、自分の演技で映画の成功を左右できるという意味で、槌子を手に入れている。

CEOs also have both measurement and leverage. They're measured, in that the performance of the company is their performance. And they have leverage in that their decisions set the whole company moving in one direction or another.

CEO もまた、測定と槌子を持っている。会社の成績が自分の成績になるという意味で測られるし、自分の決断によって会社全ての方向が決まるという意味で槌子を持っている。

I think everyone who gets rich by their own efforts will be found to be in a situation with measurement and leverage. Everyone I can think of does: CEOs, movie stars, hedge fund managers, professional athletes. A good hint to the presence of leverage is the possibility of failure. Upside must be balanced by downside, so if there is big potential for gain there must also be a terrifying possibility of loss. CEOs, stars, fund managers, and athletes all live with the sword hanging over their heads; the moment they start to suck, they're out. If you're in a job that feels safe, you are not going to get rich, because if there is no danger there is almost certainly no leverage.

自分の努力によって裕福になった人々は誰もが、測定と槌子のある環境にいたのだと思う。私が思い付く全ての人はそうだ。CEO、映画スター、ヘッジファンドのマネージャ、プロスポーツ選手。槌子の存在を知る良いヒントは、リスクの可能性だ。良いことは悪いこととバランスしていなければならない。大きな利益を得られる可能性がある場所には、常に恐ろしいリスクの危険も潜んでいる。CEO、スター、ファンドマネージャ、スポーツ選手は常に、自分の頭上に剣が吊るされているような人生を送っている。一度でも失敗すればそれで終わりだ。安心できるような仕事に就いている限り、裕福になれる可能性はないだろう。リスクのない場所には、まず槌子が存在しないからだ。

But you don't have to become a CEO or a movie star to be in a situation with measurement and leverage. All you need to do is be part of a small group working on a hard problem.

でも、測定と槌子を手に入れるために、CEO や映画スターになる必要はない。難しい問題に取り組む小さな集団に参加すればいいんだ。

Smallness = Measurement

小ささ＝測定

If you can't measure the value of the work done by individual employees, you can get close. You can measure the value of the work done by small groups.

従業員一人一人の仕事の価値は測れないとしても、近いことはできる。小さなグループによる仕事の価値は測れるんだ。

One level at which you can accurately measure the revenue generated by employees is at the level of the whole company. When the company is small, you are thereby fairly close to measuring the contributions of individual employees. A viable startup might only have ten employees, which puts you within a factor of ten of measuring individual effort.

会社全体のレベルでは、従業員たちの生み出した利益を正確に測れる。したがって、会社が小さければ、各人の貢献を測ることに近いところまで行ける。元気なベンチャー企業は10人くらいしか従業員がいないかもしれないが、それは各人の努力が、10倍くらいの係数の範囲で測れるということを意味する。

Starting or joining a startup is thus as close as most people can get to saying to one's boss, I want to work ten times as hard, so please pay me ten times as much. There are two differences: you're not saying it to your boss, but directly to the customers (for whom your boss is only a proxy after all), and you're not doing it individually, but along with a small group of other ambitious people.

ベンチャー企業を興したり参加したりすることは、多くの人々にとっては、ちょうど上司に10倍働くので10倍給料をくれということに一番近い方法だろう。ただ、2つの違いがある。まず、それを言う相手は上司ではなく直接の顧客であるということ(結局、上司というのは顧客の代理でしかないんだ)。それに、自分一人でそう言うのではなく、少数の野心的な仲間と一緒にやるってことだ。

It will, ordinarily, be a group. Except in a few unusual kinds of work, like acting or writing books, you can't be a company of one person. And the people you work with had better be good, because it's their work that yours is going to be averaged with.

通常、それはグループになるだろう。俳優とか小説家のような例外的な仕事を除いては、会社を一人で運営してゆくことはできない。そして、一緒に何かをしようという人は優秀でなければならない。だって自分の仕事は彼らの仕事と平均を取られるわけだから。

A big company is like a giant galley driven by a thousand rowers. Two things keep the speed of the galley down. One is that individual rowers don't see any result from working harder. The other is that, in a group of a thousand people, the average rower is likely to be pretty average.

大企業は 1000 人の漕ぎ手がいる巨大なガレー船のようなものだ。この船が遅くなる理由が 2 つある。まず、個々の漕ぎ手が自分の努力の結果を見ることができないこと。そして、1000 人も人がいれば、平均的な漕ぎ手というのは文字通り平均的である可能性が高いことだ。

If you took ten people at random out of the big galley and put them in a boat by themselves, they could probably go faster. They would have both carrot and stick to motivate them. An energetic rower would be encouraged by the thought that he could have a visible effect on the speed of the boat. And if someone was lazy, the others would be more likely to notice and complain.

その船からランダムに 10 人の漕ぎ手を選んで別の船に乗せれば、たぶんその船は速く進むだろう。目の前のエンジンがぶら下がっているからだ。力が余っている漕ぎ手は、自分の努力次第で船のスピードを上げられると知ればやる気を出すだろう。そして誰かが怠けていたら他の人は気付いて文句を言うだろう。

But the real advantage of the ten-man boat shows when you take the ten *best* rowers out of the big galley and put them in a boat together. They will have all the extra motivation that comes from being in a small group. But more importantly, by selecting that small a group you can get the best rowers. Each one will be in the top 1%. It's a much better deal for them to average their work together with a small group of their peers than to average it with everyone.

だがこの 10 人の船が本当に力を発揮するのは、巨大なガレー船の漕ぎ手のうち最も優秀な 10 人を選んで乗せたときだ。もちろん彼らは小さなグループにいることによってさらにやる気を出すだろうが、より重要なのは、小さなグループを選ぶことにより優秀な人物を選べるということだ。トップ 1%に入る漕ぎ手で構成できるんだ。全員で平均を取るより少数精鋭で平均をとったほうがいいに決まっている。

That's the real point of startups. Ideally, you are getting together with a group of other people who also want to work a lot harder, and get paid a lot more, than they would in a big company. And because startups tend to get founded by self-selecting groups of ambitious people who already know one another (at least by reputation), the level of measurement is more precise than you get from smallness alone. A startup is not merely ten people, but ten people like you.

これが、ベンチャー企業の本当の意味だ。理想的には、大企業でやるよりずっとたくさん働いて、ずっとたくさん給料をもらいたいという集団を集めることができる。しかも多くの場合、ベンチャー企業の創業者たちは、お互い既に知っている(か、少なくとも評判を聞

いている)、野心的で自分から何かをしようとする集団になるから、ただの小集団よりもずっと測定は正確になる。ベンチャー企業はただ 10 人集まっただけじゃない。10 人のあなたのような人物が集まった集団なんだ。

Steve Jobs once said that the success or failure of a startup depends on the first ten employees. I agree. If anything, it's more like the first five. Being small is not, in itself, what makes startups kick butt, but rather that small groups can be select. You don't want small in the sense of a village, but small in the sense of an all-star team.

スティーブ・ジョブズは、ベンチャー企業の成功と失敗は最初の 10 人の従業員で決まると言ったことがある。私もそれに同意する。もし言い換えられるなら、最初の 5 人と言ってもいいだろう。ベンチャー企業をすごい場所にするのは、ただ小さくあるだけじゃなくて、選り抜かれた小さい集団であることだ。村のような意味で小さい集団でなく、オールスターチームのような集団が必要なんだ。

The larger a group, the closer its average member will be to the average for the population as a whole. So all other things being equal, a very able person in a big company is probably getting a bad deal, because his performance is dragged down by the overall lower performance of the others. Of course, all other things often are not equal: the able person may not care about money, or may prefer the stability of a large company. But a very able person who does care about money will ordinarily do better to go off and work with a small group of peers.

集団が大きくなればなるほどその平均は人口全体の平均に近づいてゆく。したがって、他の条件が全て同じなら、大企業にいる非常に能力の高い人は損をしていることになる。彼の成果は他のより低い成果に足を引きずられてしまうからだ。もちろん他の条件はいつも同じではない。お金を儲けることに関心がない人だっているし、大企業の安定性を重視する人だっている。だが、お金に関心があり、かつ能力のある人は、同じような仲間との小さな集団で働くほうがよいだろう。

Technology = Leverage

技術＝槌子

Startups offer anyone a way to be in a situation with measurement and leverage. They allow measurement because they're small, and they offer leverage because they make money by inventing new technology.

ベンチャー企業は測定と槌子のある環境を提供してくれる。小さいことで測定が可能にな

り、新しい技術を発明してお金を儲けることが梃子になる。

What is technology? It's *technique*. It's the way we all do things. And when you discover a new way to do things, its value is multiplied by all the people who use it. It is the proverbial fishing rod, rather than the fish. That's the difference between a startup and a restaurant or a barber shop. You fry eggs or cut hair one customer at a time. Whereas if you solve a technical problem that a lot of people care about, you help everyone who uses your solution. That's leverage.

技術とは何だろうか。それは手法だ。何かを行う際の方法だ。新しい方法で何かをやれることを見つけたとすれば、他の皆が同じ方法を使うことで、その価値は人数分増えることになる。よく言われるように、それは魚ではなく釣竿なんだ。ベンチャー企業がレストランや美容室と違うのはそこだ。そういう店では、一個ずつ卵を炒めたり、一人ずつお客の髪の毛を切る。しかし、多くの人々が関心を持つ技術上の問題を解けば、その解法を使う全ての人を助けられる。これが梃子だ。

If you look at history, it seems that most people who got rich by creating wealth did it by developing new technology. You just can't fry eggs or cut hair fast enough. What made the Florentines rich in 1200 was the discovery of new techniques for making the high-tech product of the time, fine woven cloth. What made the Dutch rich in 1600 was the discovery of shipbuilding and navigation techniques that enabled them to dominate the seas of the Far East.

歴史を見れば、富を創ることで裕福になった人々の多くは、新しい技術を創ることでそうしたと分かるだろう。卵を素早く炒めたり髪を素早く切るだけでは不十分なんだ。1200年のフィレンツェを裕福にしたのは、当時のハイテク製品である毛織物を生産する新しい手法だった。1600年のオランダを裕福にしたのは新しい造船技術と航海術で、それにより彼らは極東までの航路を一手に収めた。

Fortunately there is a natural fit between smallness and solving hard problems. The leading edge of technology moves fast. Technology that's valuable today could be worthless in a couple years. Small companies are more at home in this world, because they don't have layers of bureaucracy to slow them down. Also, technical advances tend to come from unorthodox approaches, and small companies are less constrained by convention.

幸いなことに、小さな集団は困難な問題を解くことに適している。技術の最先端は非常に速く動いている。今日価値のある技術でも、2年後には価値がなくなっているかもしれない。小さな会社はこのような世界を心地良く感じる。動きを止める、層になった官僚的なシス

テムを持たないからだ。また、技術の進歩はしばしば突拍子もない挑戦から生まれるが、小さな会社はそこでも伝統に縛られないという利点がある。

Big companies can develop technology. They just can't do it quickly. Their size makes them slow and prevents them from rewarding employees for the extraordinary effort required. So in practice big companies only get to develop technology in fields where large capital requirements prevent startups from competing with them, like microprocessors, power plants, or passenger aircraft. And even in those fields they depend heavily on startups for components and ideas.

大企業も技術を開発することはできる。ただそれを素早く行うことはできない。その大きさのために、進行は遅くなり、並々ならぬ努力を求められた従業員に報いることが難しくなる。結局、大企業が技術を開発するのは、多大な投資が必要なためにベンチャー企業が参入しにくい分野に限られる。マイクロプロセッサ、発電所、あるいは旅客機といったものだ。しかもこれらの分野でさえ、個々の要素技術やアイディアはベンチャー企業に負うことが多い。

It's obvious that biotech or software startups exist to solve hard technical problems, but I think it will also be found to be true in businesses that don't seem to be about technology. McDonald's, for example, grew big by designing a system, the McDonald's franchise, that could then be reproduced at will all over the face of the earth. A McDonald's franchise is controlled by rules so precise that it is practically a piece of software. Write once, run everywhere. Ditto for Wal-Mart. Sam Walton got rich not by being a retailer, but by designing a new kind of store. バイオテクノロジーやソフトウェアで困難な技術的問題を解くベンチャー企業が出てくるのは当然だが、この原理は技術とは一見関係ないビジネスでも正しいと思う。例えばマクドナルドは、フランチャイズというシステムを設計し、それを世界中にコピーすることで大きくなった。マクドナルドのフランチャイズは非常に綿密な規則により制御されていて、まるでソフトウェアみたいだ。「一度書けばどこでも走る」というやつだ。ウォルマートも同じだ。サム・ウォルトンは小売店をやることで裕福になったんじゃない。新しい種類の店を設計することでそうなったんだ。

Use difficulty as a guide not just in selecting the overall aim of your company, but also at decision points along the way. At Viaweb one of our rules of thumb was *run upstairs*. Suppose you are a little, nimble guy being chased by a big, fat, bully. You open a door and find yourself in a staircase. Do you go up or down? I say up. The

bully can probably run downstairs as fast as you can. Going upstairs his bulk will be more of a disadvantage. Running upstairs is hard for you but even harder for him.

困難な問題を選ぶというのは、最初に会社の目標を選ぶときに役立つだけじゃない。業務の中で決断を下さなければならないときの指針にもなる。Viawebでの第一の規則は「階段を駆け上がれ」だった。あなたが小さく非力な男で、巨大で太ったいじめっ子に追いかけているとしたら、上へ逃げる?それとも下へ逃げる?私は上を選ぶ。いじめっ子はたぶんあなたと同じくらい速く階段を駆け下りられるだろう。でも駆け上がる時、いじめっ子の体重は負荷になる。もちろんあなただって駆け上がるのは苦しい。でも相手のほうがもっと苦しいんだ。

What this meant in practice was that we deliberately sought hard problems. If there were two features we could add to our software, both equally valuable in proportion to their difficulty, we'd always take the harder one. Not just because it was more valuable, but *because it was harder*. We delighted in forcing bigger, slower competitors to follow us over difficult ground. Like guerillas, startups prefer the difficult terrain of the mountains, where the troops of the central government can't follow. I can remember times when we were just exhausted after wrestling all day with some horrible technical problem. And I'd be delighted, because something that was hard for us would be impossible for our competitors. 現実の場面でこのたとえが意味するのは、私たちは意識して難しい問題を探したということだ。もし私たちがソフトウェアに追加すべき機能を 2 つのうちから選ばなくてはならなくて、どちらもその困難度に比例した価値があったとしたら、常に難しいほうを選んだ。そちらのほうが価値が高いというだけでなく、そちらのほうが難しいという理由からだ。それによって、より大きく動きの遅い競争相手が、より困難な土俵で私たちを追いかけるのに苦労しているのを見て喜んだものだ。ベンチャー企業は、ゲリラのように、中央政府軍が追ってこられないような険しい山岳地帯を好む。私は、おそろしく困難な技術上の問題にまる一日がかりで皆で取り組み、精も根も尽き果ててしまった時のことを覚えている。それでも私は嬉しかった。私たちがこれだけ苦労するような問題なら、きっと競争相手は誰も解決できないだろうと思ったからだ。

This is not just a good way to run a startup. It's what a startup is. Venture capitalists know about this and have a phrase for it: *barriers to entry*. If you go to a VC with a new idea and ask him to invest in it, one of the first things he'll ask is, how hard would this be for someone else to develop? That is, how much difficult ground have you put between yourself and potential pursuers? [7] And you had

better have a convincing explanation of why your technology would be hard to duplicate. Otherwise as soon as some big company becomes aware of it, they'll make their own, and with their brand name, capital, and distribution clout, they'll take away your market overnight. You'd be like guerillas caught in the open field by regular army forces.

これは、ベンチャー企業を経営していくのに良い方法だというだけじゃない。そもそもこういうことをするのがベンチャー企業なんだ。ベンチャーキャピタリストはもちろんこのことを知っていて、それに対する用語まである。「参入障壁」だ。あなたが新しいアイデアを思い付いてベンチャーキャピタルに投資をもちかけたとしたら、最初に尋ねられるのはそれを他の誰かが作るのはどれだけ難しいか、ということだ。言い換えれば、あなたと他の追跡者との間にどれだけ困難な領域が確保してあるか、だ。[\[7\]](#) そして、あなたはその技術を他で作るのがどれだけ困難かという、説得力のある説明を用意しておかなければならない。でなければ、大会社はそのアイデアを見た途端に自分で同じものを作り、大会社のブランド名、資本、販売チャネルを使って、あなたの市場を一夜にして奪ってしまうだろう。まさに、正規軍に平原の真ん中で捕まるゲリラみたいなものだ。

One way to put up barriers to entry is through patents. But patents may not provide much protection. Competitors commonly find ways to work around a patent. And if they can't, they may simply violate it and invite you to sue them. A big company is not afraid to be sued; it's an everyday thing for them. They'll make sure that suing them is expensive and takes a long time. Ever heard of Philo Farnsworth? He invented television. The reason you've never heard of him is that his company was not the one to make money from it. [\[8\]](#) The company that did was RCA, and Farnsworth's reward for his efforts was a decade of patent litigation. 参入障壁を作るひとつの方法は特許によるものだ。しかし、特許は大した障壁にはならないかもしれない。競争相手はうまく特許を回避する方法を見つけるからだ。たとえそれができなくても、わざと特許を侵害して裁判に持ち込む戦術を取るかもしれない。大会社は訴えられることを恐れない。日常的なことだからだ。彼らは確実に裁判を長期化させ、こちらは莫大な費用を払わされることになる。フィロ・ファーンズワースの話聞いたことはあるかい。彼はテレビを発明したんだ。彼のことなど普通の人は知らないだろうが、それは彼の会社がテレビでお金を儲けることがなかったからだ。[\[8\]](#) テレビで金を儲けたのは RCA であり、ファーンズワースが得た見返りといえば、十年にもわたる特許裁判の苦労だけだった。

Here, as so often, the best defense is a good offense. If you can develop technology that's simply too hard for competitors to duplicate, you don't need to

rely on other defenses. Start by picking a hard problem, and then at every decision point, take the harder choice. [9]

ここでも、攻撃こそが最大の防御になる。競争相手が真似するのが難し過ぎるような技術を作りさえすれば、他の防御に頼る必要はない。難しい問題を選ぶことから始め、決断が必要な場面では常に難しいほうを選べばよい。[9]

The Catch(es)

落とし穴

If it were simply a matter of working harder than an ordinary employee and getting paid proportionately, it would obviously be a good deal to start a startup. Up to a point it would be more fun. I don't think many people like the slow pace of big companies, the interminable meetings, the water-cooler conversations, the clueless middle managers, and so on.

普通の会社員をやっているより頑張って働けばそれに比した給料が貰えるというだけなら、ベンチャーを立ち上げるのは何も損のないことだ。それだけなら、ベンチャーのほうが面白いだろうし。多くの人は、大企業のゆっくりとしたペース、いつ終わるともしれない会議、休憩室でのおしゃべり、物分かりの悪い中間管理職とかいったものにうんざりしているんじゃないかと思う。

Unfortunately there are a couple catches. One is that you can't choose the point on the curve that you want to inhabit. You can't decide, for example, that you'd like to work just two or three times as hard, and get paid that much more. When you're running a startup, your competitors decide how hard you work. And they pretty much all make the same decision: as hard as you possibly can.

残念ながらここには落とし穴も 2 つばかりある。ひとつは、曲線上のどの点に留まるかを自分で選ぶことができないということだ。例えば、どれくらい働きたいのかを自分で決めることはできない。だいたい 2 倍か 3 倍働いてその分給料を貰いたいな、というわけにはいかない。ベンチャーを立ち上げたら、競争相手があなたがどれだけ働かなくちゃならないかを決めることになるんだ。そしてどの会社も同じような結論へと向かう。つまり、可能な限り頑張るってことだ。

The other catch is that the payoff is only on average proportionate to your productivity. There is, as I said before, a large random multiplier in the success of any company. So in practice the deal is not that you're 30 times as productive and get paid 30 times as much. It is that you're 30 times as productive, and get paid

between zero and a thousand times as much. If the mean is 30x, the median is probably zero. Most startups tank, and not just the dogfood portals we all heard about during the Internet Bubble. It's common for a startup to be developing a genuinely good product, take slightly too long to do it, run out of money, and have to shut down.

もうひとつの落とし穴は、自分の生産性に比例した見返りというのは平均値にすぎないということだ。先に述べたように、どんな会社の成功にも大きくランダムな係数がかかってくる。したがって現実には、30 倍働いたから 30 倍貰えるということにはならない。あなたが 30 倍働いたとしたら、貰える額はおそらくゼロから 1000 倍の間のどこかになるだろう。そして、平均値が 30 倍だとしても中央値はたぶんゼロだ。多くのベンチャー企業は生き残れない。インターネットバブルのころによく目にしたドッグフードポータルのようなものだけじゃない。本当に良い製品を作っているのに、少し長くかかりすぎただけで、資本を使い果たし潰れてしまうベンチャー企業は珍しくはない。

A startup is like a mosquito. A bear can absorb a hit and a crab is armored against one, but a mosquito is designed for one thing: to score. No energy is wasted on defense. The defense of mosquitos, as a species, is that there are a lot of them, but this is little consolation to the individual mosquito.

ベンチャー企業は蚊みたいなものだ。熊なら叩かれてもけろっとしているだろうし、蟹なら叩かれても硬い甲羅で身を守るが、蚊はただひとつのこゝのためだけに作られている。刺すことだ。防御に対しては一切エネルギーが使われていない。種としては、蚊はたくさんいることが防御となっているが、それは一匹一匹の蚊にとってはあまり慰めにはならないだろう。

Startups, like mosquitos, tend to be an all-or-nothing proposition. And you don't generally know which of the two you're going to get till the last minute. Viaweb came close to tanking several times. Our trajectory was like a sine wave. Fortunately we got bought at the top of the cycle, but it was damned close. While we were visiting Yahoo in California to talk about selling the company to them, we had to borrow a conference room to reassure an investor who was about to back out of a new round of funding that we needed to stay alive.

ベンチャー企業は、蚊と同じように、オール・オア・ナッシングの立場に立つことが多い。そして最後の瞬間までどちらに転ぶかは分からないものだ。Viaweb にも何度も危ない場面があった。私たちの軌跡はまるでサイン曲線のようなものだった。幸いにして曲線の頂点で買収されることができたが、本当にぎりぎりだったんだ。カリフォルニアに Yahoo を訪問して会社の買収の交渉をしていたのと並行して、私たちは会議室を借りて会社存続を賭

けた次のラウンドの資本投資を確実にしてもらうよう投資家たちに掛け合っていたんだ。

The all-or-nothing aspect of startups was not something we wanted. Viaweb's hackers were all extremely risk-averse. If there had been some way just to work super hard and get paid for it, without having a lottery mixed in, we would have been delighted. We would have much preferred a 100% chance of \$1 million to a 20% chance of \$10 million, even though theoretically the second is worth twice as much. Unfortunately, there is not currently any space in the business world where you can get the first deal.

この、のるかそるかという状態は、決して嬉しいものじゃない。Viaweb のハッカーたちは特にリスクを回避したがる質だった。ものすごく働いてそれに見合うものをもらうだけでいい方法、くじの要素が入ってこない方法があったなら、狂喜していたことだろう。20% のチャンスで 1000 万ドル得るよりも、100% のチャンスで 100 万ドルを得ることができれば、理論的には前者のほうが 2 倍期待値が高いとしても、後者を選んでいただろう。残念ながら、現在のビジネスの世界のどこを探しても後者のような方法は存在しない。

The closest you can get is by selling your startup in the early stages, giving up upside (and risk) for a smaller but guaranteed payoff. We had a chance to do this, and stupidly, as we then thought, let it slip by. After that we became comically eager to sell. For the next year or so, if anyone expressed the slightest curiosity about Viaweb we would try to sell them the company. But there were no takers, so we had to keep going.

できることと言えば、ベンチャー企業をなるべく早い段階で売ってしまい、得られたかもしれない巨大な利益(とリスク)の代わりに、保証された小さな見返りを得ることだろう。私たちにもそういうチャンスが何度か巡ってきたのだが、愚かなことに(と当時の私たちは考えていた)それを見逃してしまった。その後で、私たちはおかしいほど売ることに熱心になった。しかしその翌年から翌々年にかけては、誰も Viaweb なんて鼻にもかけなかった。会社を売ろうとしても誰も買ってくれなかったから、続けていくしかなかったんだ。

It would have been a bargain to buy us at an early stage, but companies doing acquisitions are not looking for bargains. A company big enough to acquire startups will be big enough to be fairly conservative, and within the company the people in charge of acquisitions will be among the more conservative, because they are likely to be business school types who joined the company late. They would rather overpay for a safe choice. So it is easier to sell an established startup, even at a large premium, than an early-stage one.

私たちが初期のうちに買っていれば、それは安い買い物になったろう。でも、買収をするような企業は、安い買い物を探しているわけじゃない。ベンチャー企業を買収できる規模の企業というのは、通常は保守的にならざるを得ないくらい大規模で、しかも買収の責任者となるような人々の中でも最も保守的な、ビジネススクールを出て後からその会社に就職したような人であることが多い。安全な選択ができるなら払いすぎてもよいとさえ考える人々だ。だから、初期のベンチャー企業よりは、たとえ金額が大きくても、既に確立したベンチャー企業を売るほうが易しいんだ。

Get Users

ユーザを獲得すること

I think it's a good idea to get bought, if you can. Running a business is different from growing one. It is just as well to let a big company take over once you reach cruising altitude. It's also financially wiser, because selling allows you to diversify. What would you think of a financial advisor who put all his client's assets into one volatile stock?

可能なら、買収されることは良いことだと思う。企業を継続して経営してゆくことは、育てることとは違う。巡航高度に達したら大企業に後を任せるのがよいだろう。またそれは財政的にも有利だ。売ることでもリスクを分散できる。ファイナンシャルアドバイザーが、顧客の資産を不安定な株 1 銘柄に全て突っ込んだらどう思う？

How do you get bought? Mostly by doing the same things you'd do if you didn't intend to sell the company. Being profitable, for example. But getting bought is also an art in its own right, and one that we spent a lot of time trying to master. どうやったら買収してもらえるだろう。やることは、買収を目標にしない場合とたいして変わりはない。例えば利益を出すとかね。でも、買収されることにはそれ特有の技術とものがあり、私たちもそれをマスターするのに長い時間をかけたものだ。

Potential buyers will always delay if they can. The hard part about getting bought is getting them to act. For most people, the most powerful motivator is not the hope of gain, but the fear of loss. For potential acquirers, the most powerful motivator is the prospect that one of their competitors will buy you. This, as we found, causes CEOs to take red-eyes. The second biggest is the worry that, if they don't buy you now, you'll continue to grow rapidly and will cost more to acquire later, or even become a competitor.

買い取ってくれそうな会社は、可能ならいつだって決断を遅らせようとする。買い取られ

るに当たって難しいのは、彼らの腰を上げさせることだ。多くの人々にとって一番強い動機付けは、利益への期待ではなく、損失への恐れだ。買い取り会社候補にとっての一番の動機付けは、彼らの競争相手が買ってしまうかもしれないという恐れなんだ。これがあるから CEO は深夜のフライトに飛び乗ってくる、ということを私たちは学んだ。二番目の心配は、もし今のうちに買っておかなければもっと急速にあなたの会社が大きくなって、後で買うにはもっとずっと金がかかるか、下手をすると競争相手にさえなるかもしれないと思わせることだ。

In both cases, what it all comes down to is users. You'd think that a company about to buy you would do a lot of research and decide for themselves how valuable your technology was. Not at all. What they go by is the number of users you have.

どちらの場合でも、一番大事なところは、結局ユーザに行き着く。ベンチャー企業を買収しようなんて会社は、たくさんの調査をしてターゲットの技術がどれくらい価値があるかを決めようとする、と思うかもしれない。全然そんなことはないんだ。彼らが聞くのは、あなたが持っているユーザ数だ。

In effect, acquirers assume the customers know who has the best technology. And this is not as stupid as it sounds. Users are the only real proof that you've created wealth. Wealth is what people want, and if people aren't using your software, maybe it's not just because you're bad at marketing. Maybe it's because you haven't made what they want.

実質的に、買収しようとしている企業は、顧客こそが一番の技術を知っていると仮定していることになる。これはそれほどおかしい考えではない。ユーザはあなたが富を創り出したことの唯一の現実的な証明だからだ。富は人々が欲しがるものであり、人々があなたのソフトウェアを使っていないのなら、それはマーケティングが下手なだけじゃないかもしれない。それは、あなたが作ったものは人々が欲しがるものではなかったということなのかもしれないんだ。

Venture capitalists have a list of danger signs to watch out for. Near the top is the company run by techno-weenies who are obsessed with solving interesting technical problems, instead of making users happy. In a startup, you're not just trying to solve problems. You're trying to solve problems *that users care about*. ベンチャーキャピタルは、注意すべき危険信号のリストを持っている。トップ近くにあるのは、ユーザを幸せにすることより面白い技術的問題を解くことのみに夢中な、技術馬鹿によって会社が経営されている場合だ。ベンチャー企業は単に問題を解くだけじゃいけな

い。ユーザが解いてほしいと思っている問題を解くべきなんだ。

So I think you should make users the test, just as acquirers do. Treat a startup as an optimization problem in which performance is measured by number of users. As anyone who has tried to optimize software knows, the key is measurement. When you try to guess where your program is slow, and what would make it faster, you almost always guess wrong.

したがって、買収しようとする企業と同じように、あなたもユーザを指標とするべきだと思う。ベンチャー企業を最適化の問題と考えてみよう。その性能がユーザ数で測られるものとするんだ。ソフトウェアを最適化した経験のある人なら誰もが知っているように、鍵は測定にある。ソフトウェアのどの部分が遅くて、どこをどうしたら速くなるかを推測に頼っているときは、その推測はまず確実に間違っている。

Number of users may not be the perfect test, but it will be very close. It's what acquirers care about. It's what revenues depend on. It's what makes competitors unhappy. It's what impresses reporters, and potential new users. Certainly it's a better test than your a priori notions of what problems are important to solve, no matter how technically adept you are.

ユーザ数は完全なテストではないにしても、かなり近いものにはなるだろう。買収しようとする企業はそれを気にしている。収益もユーザ数に依存している。ユーザ数が多ければライバルは心配する。記者は関心し、新しいユーザ候補も関心を持つだろう。確かにこれは、どの問題を解くのが重要なのかを頭から決めてかかるよりも良い指標だ。たとえあなたが技術的にどんなに優れていようとも。

Among other things, treating a startup as an optimization problem will help you avoid another pitfall that VCs worry about, and rightly-- taking a long time to develop a product. Now we can recognize this as something hackers already know to avoid: premature optimization. Get a version 1.0 out there as soon as you can. Until you have some users to measure, you're optimizing based on guesses.

他にも、ベンチャー企業を最適化問題と考えれば、ベンチャーキャピタルが恐れる別の危険を避けやすくなる。製品開発に長く時間をかけ過ぎるという危険だ。この危険は、ハッカーに既に知られている、別の避けるべき問題と同じだ。早過ぎる最適化というやつだ。バージョン 1.0 を可能な限り速く出そう。ユーザを獲得しない限り、手探りで最適化からは逃れられない。

The ball you need to keep your eye on here is the underlying principle that wealth

is what people want. If you plan to get rich by creating wealth, you have to know what people want. So few businesses really pay attention to making customers happy. How often do you walk into a store, or call a company on the phone, with a feeling of dread in the back of your mind? When you hear "your call is important to us, please stay on the line," do you think, oh good, now everything will be all right?

常に注意しておかねばならないことは、富とは人々が欲するものであるという基本原理だ。富を作り出すことで裕福になろうとしているなら、人々が欲するものは何かを知っていなければならない。顧客を幸せにすることに本当に注意を払っているビジネスはほとんど存在しない。内心びくびくしながら、店に入ったり、会社に電話をかけたりすることは多いだろう。電話で「あなたの電話は私たちにとって重要です。どうかしばらくお待ちください」という録音されたアナウンスを聞いて、うん、いいぞ、これで大丈夫だ、なんて思ukai?

A restaurant can afford to serve the occasional burnt dinner. But in technology, you cook one thing and that's what everyone eats. So any difference between what people want and what you deliver is multiplied. You please or annoy customers wholesale. The closer you can get to what they want, the more wealth you generate.

レストランは、たまには焦げた料理を出しても何とかやっていける。でも技術の分野では、ひとつの料理を皆が食べるんだ。人々が欲するものとあなたが作るもののギャップは、何倍にも拡大される。顧客全員を喜ばせるか、煩わせるかだ。顧客が欲しがるものに近づけば近づくほど、生み出せる富は大きくなる。

Wealth and Power

富と力

Making wealth is not the only way to get rich. For most of human history it has not even been the most common. Until a few centuries ago, the main sources of wealth were mines, slaves and serfs, land, and cattle, and the only ways to acquire these rapidly were by inheritance, marriage, conquest, or confiscation. Naturally wealth had a bad reputation.

富を創ることは裕福になる唯一の方法ではない。人類の歴史の大部分では、むしろ他の方法がずっと多かった。数世紀前までは、富の主要な源は鉱山や農奴、土地、あるいは家畜であり、それらを一気に得るには、相続するか、結婚するか、征服するか、没収するかしかなかった。富に悪い評判が立つのも自然なことだ。

Two things changed. The first was the rule of law. For most of the world's history, if you did somehow accumulate a fortune, the ruler or his henchmen would find a way to steal it. But in medieval Europe something new happened. A new class of merchants and manufacturers began to collect in towns. [10] Together they were able to withstand the local feudal lord. So for the first time in our history, the bullies stopped stealing the nerds' lunch money. This was naturally a great incentive, and possibly indeed the main cause of the second big change, industrialization.

2つの変化があった。まず、法律による規制だ。世界の歴史の大部分の期間では、誰かが財を成すと、支配者やその取り巻きたちはそれを掠めとる方法を見出した。しかし、中世のヨーロッパで新しいことが起きたんだ。新しい階級である商人や工場主が都市に集まり出した。[10] 彼らは一緒になって、地域の領主に立ち向かった。そして歴史上初めて、いじめっ子がいじめられっ子の弁当を盗むのを止めさせることができたんだ。これはもちろん、第二の変化である工業化への大きな動機付けになった。いや、むしろ主要な理由であったかもしれない。

A great deal has been written about the causes of the Industrial Revolution. But surely a necessary, if not sufficient, condition was that people who made fortunes be able to enjoy them in peace. [11] One piece of evidence is what happened to countries that tried to return to the old model, like the Soviet Union, and to a lesser extent Britain under the labor governments of the 1960s and early 1970s. Take away the incentive of wealth, and technical innovation grinds to a halt.

産業革命の理由については既にさまざまな論説が書かれている。だが、自分の作った財産を平和に持つておけるということは、十分条件でないにせよ、必要条件ではあったはずだ。

[11] 証拠のひとつは、ソビエト連邦や、英国の1960年代から1970年代の労働党政権のように、古いモデルに戻ろうとした国に何が起こったかだ。富への動機付けを取ってしまえば、技術革新はほとんど停止してしまう。

Remember what a startup is, economically: a way of saying, I want to work faster. Instead of accumulating money slowly by being paid a regular wage for fifty years, I want to get it over with as soon as possible. So governments that forbid you to accumulate wealth are in effect decreeing that you work slowly. They're willing to let you earn \$3 million over fifty years, but they're not willing to let you work so hard that you can do it in two. They are like the corporate boss that you can't go to and say, I want to work ten times as hard, so please pay me ten times as much.

Except this is not a boss you can escape by starting your own company.

経済的に、ベンチャー企業が何を意味していたかを思い出してほしい。それは「私はもっと働きたい」と主張する手段だった。50 年間、普通の給料をゆっくり貯める代わりに、できる限り早くお金を得たい、と。もし政府が富を貯めることを禁じるなら、それは実質的に、ゆっくり働けと命じているに等しい。50 年の間に 300 万ドル稼ぐことは推奨するけれど、ものすごく働くことで 2 年間でそれをやるのはダメ、というわけだ。ちょうど、10 倍働くから 10 倍給料をくれと言っても通じない、大企業の上司みたいなものだ。違いは、国の場合はそこから抜けて新しい会社を始めるわけにいかないことだ。

The problem with working slowly is not just that technical innovation happens slowly. It's that it tends not to happen at all. It's only when you're deliberately looking for hard problems, as a way to use speed to the greatest advantage, that you take on this kind of project. Developing new technology is a pain in the ass. It is, as Edison said, one percent inspiration and ninety-nine percent perspiration. Without the incentive of wealth, no one wants to do it. Engineers will work on sexy projects like fighter planes and moon rockets for ordinary salaries, but more mundane technologies like light bulbs or semiconductors have to be developed by entrepreneurs.

ゆっくり働くことの問題は、技術革新が遅くなるだけじゃない。革新が全く起こらなくなることなんだ。革新を起こすようなプロジェクトをやりたいと思うのは、スピードを味方にして敢えて難しい問題に挑戦しようとするときに限られる。新しい技術を開発するのはものすごく大変なことだ。エジソンが言ったように、1%のひらめきと、99%の努力なんだ。富への動機付けがなければ、誰もそんなことをやろうとはしないだろう。エンジニアは確かに、月ロケットとか戦闘機みたいな華々しいプロジェクトなら普通の給料でも仕事をするだろうけれど、もっと地味な技術、電球や半導体のようなものは、起業家によって開発されなければならない。

Startups are not just something that happened in Silicon Valley in the last couple decades. Since it became possible to get rich by creating wealth, everyone who has done it has used essentially the same recipe: measurement and leverage, where measurement comes from working with a small group, and leverage from developing new techniques. The recipe was the same in Florence in 1200 as it is in Santa Clara today.

ベンチャー企業は、ここ 20 年の間にシリコンバレーに出てきたものじゃない。富を創り出すことで裕福になることが可能になってからというもの、それをした全ての人が、同じレシピを使ってきたんだ。測定と槌子だ。小さな集団で働くことで測定を得て、新しい技術

を開発することで梃子を得る。このレシピは、1200年のフィレンツェでも、今日のサンタクララでも、何の違いもない。

Understanding this may help to answer an important question: why Europe grew so powerful. Was it something about the geography of Europe? Was it that Europeans are somehow racially superior? Was it their religion? The answer (or at least the proximate cause) may be that the Europeans rode on the crest of a powerful new idea: allowing those who made a lot of money to keep it.

このことを理解することで、ヨーロッパはなぜあんなに成長したのか、という重要な問いの答えが見つかる。ヨーロッパの地理が重要だったのだろうか。それともヨーロッパ人が人種的に優れていたのだろうか。それとも宗教のせいかな。答えは(少なくとも近似的には)、財を成した人にそれを持っていたとしてもよいとする、という強力で新しいアイディアの波に乗ったからだ。

Once you're allowed to do that, people who want to get rich can do it by generating wealth instead of stealing it. The resulting technological growth translates not only into wealth but into military power. The theory that led to the stealth plane was developed by a Soviet mathematician. But because the Soviet Union didn't have a computer industry, it remained for them a theory; they didn't have hardware capable of executing the calculations fast enough to design an actual airplane.

それが許されて初めて、財を成したい人は、盗む代わりに富を生み出すようになった。この技術の進歩の成果は、富だけでなく、軍隊の力にも変化していった。ステルス戦闘機の原理となる理論はソ連の数学者により作られた。しかしソ連はコンピュータ産業を持っていなかったため、それは理論に留まった。実際の航空機を設計するのに十分なだけ速く計算ができるハードウェアを持たなかったからだ。

In that respect the Cold War teaches the same lesson as World War II and, for that matter, most wars in recent history. Don't let a ruling class of warriors and politicians squash the entrepreneurs. The same recipe that makes individuals rich makes countries powerful. Let the nerds keep their lunch money, and you rule the world.

この点では、冷戦は第二次世界大戦と同じ教訓を教えてくれるし、それを言えば近世以降のほとんどの戦争はそうだ。軍人や政治家の支配者階級に、起業家を潰させるな。個人を裕福にするのと同じレシピが、国をも裕福にする。いじめられっ子の弁当代を取り上げないでおけば、世界を征服できるんだ。

Notes

注釈

[1] One valuable thing you tend to get only in startups is *uninterruptability*. Different kinds of work have different time quanta. Someone proofreading a manuscript could probably be interrupted every fifteen minutes with little loss of productivity. But the time quantum for hacking is very long: it might take an hour just to load a problem into your head. So the cost of having someone from personnel call you about a form you forgot to fill out can be huge.

[1] ベンチャー企業のみで得られる貴重なことの一つは、中断されないことだ。職種によって、時間の単位は異なる。文章の校正係は、15 分ごとに作業を中断しても、生産性にさして影響を与えないだろう。だが、ハッキングの時間単位はとても長い。問題の全てを頭に収めるだけで 1 時間を要するかもしれない。そんな時に、あなたが書類を埋めるのを忘れたことを指摘する電話が来たりすると、その影響は計り知れない。

This is why hackers give you such a baleful stare as they turn from their screen to answer your question. Inside their heads a giant house of cards is tottering.

ハッカーが質問をされて、スクリーンから目を離す時に恐ろしく不機嫌な目つきをしているのはこのためだ。ハッカーの頭の中では、巨大なトランプの家が崩れかけているんだ。

The mere possibility of being interrupted deters hackers from starting hard projects. This is why they tend to work late at night, and why it's next to impossible to write great software in a cubicle (except late at night).

中断されるかもしれないと感じるだけで、ハッカーは難しいプロジェクトを始めるのを遅らせる。ハッカーが夜遅くに仕事をしたがるのはそのせいだ。素晴らしいソフトウェアをブースで書くことが、深夜ででもなければほとんど不可能なものもまたそうだ。

One great advantage of startups is that they don't yet have any of the people who interrupt you. There is no personnel department, and thus no form nor anyone to call you about it.

ベンチャー企業の大きな利点は、邪魔するような人がまだいないということだ。人事部はないから、書類の催促の電話もかかってこない。

[2] Faced with the idea that people working for startups might be 20 or 30 times as productive as those working for large companies, executives at large

companies will naturally wonder, how could I get the people working for me to do that? The answer is simple: pay them to.

[2] ベンチャー企業では大企業より人々が 20 倍も 30 倍も働くという事実直面して、大企業の役員は、自分のところの社員に同じくらい働いてもらうにはどうしたらよいかと頭をひねるはずだ。答えは簡単だ。その分、給料を払えばいい。

Internally most companies are run like Communist states. If you believe in free markets, why not turn your company into one?

ほとんどの企業では、内部は共産主義国家のように運営されている。自由市場をそれほどに信奉しているなら、会社もそのように運営したらどうなんだろう。

Hypothesis: A company will be maximally profitable when each employee is paid in proportion to the wealth they generate.

仮説：企業は、従業員それぞれが自分の創り出す富に比例する給料をもらえるときに、最も多くの利益を出すようになる。

[3] Until recently even governments sometimes didn't grasp the distinction between money and wealth. Adam Smith (*Wealth of Nations*, v:i) mentions several that tried to preserve their "wealth" by forbidding the export of gold or silver. But having more of the medium of exchange would not make a country richer; if you have more money chasing the same amount of material wealth, the only result is higher prices.

[3] 最近まで、政府でさえ貨幣と富の違いを把握していないことがあった。アダム・スミスは『国富論』で、「富」を保存するために金銀の輸出を禁じた国家について述べている。交換媒体をたくさん持つだけでは、国は豊かにならないのに。同じ量の富に対してより多くの貨幣があれば、単に価格が上がるだけだ。

[4] There are many senses of the word "wealth," not all of them material. I'm not trying to make a deep philosophical point here about which is the true kind. I'm writing about one specific, rather technical sense of the word "wealth." What people will give you money for. This is an interesting sort of wealth to study, because it is the kind that prevents you from starving. And what people will give you money for depends on them, not you.

[4] 「富」という言葉にはさまざまな意味があり、物質的なものだけを意味するわけじゃない。私はここで、何が本当の富かについての深い哲学的な考察をするつもりはない。むしろ、「富」という単語を、特定の技術的な意味で使っている。人々が貨幣と交換したがる

もの、という意味だ。この種の富は研究してみる価値がある。飢え死にしくなくても済むからだ。そして、人々があなたに貨幣をくれるのは、あなたにではなく、富に対してなんだ。

When you're starting a business, it's easy to slide into thinking that customers want what you do. During the Internet Bubble I talked to a woman who, because she liked the outdoors, was starting an "outdoor portal." You know what kind of business you should start if you like the outdoors? One to recover data from crashed hard disks.

ビジネスを始めるとき、あなたのやっていることを顧客が欲しているという考えに堕ちてしまうのはたやすい。インターネットバブルのころに、単に自分がアウトドアが好きだというだけで、「アウトドアポータル」を始めようとしている女性と話したことがある。アウトドアが好きなら、どういうビジネスを始めたらいいか分かるかい？クラッシュしたハードディスクからデータを復旧するサービスさ。

What's the connection? None at all. Which is precisely my point. If you want to create wealth (in the narrow technical sense of not starving) then you should be especially skeptical about any plan that centers on things you like doing. That is where your idea of what's valuable is least likely to coincide with other people's. どういう関連があるのかって？何もない。それが言いたいことだ。富を(飢え死にしないため、という狭い技術的な意味で)創り出したいのなら、自分が好きなことを中心にした計画に対しては特に疑ってかからねばならない。その分野では、何が価値があるかというあなたの判断は、他の人々と大きくずれていることが多いからだ。

[5] In the average car restoration you probably do make everyone else microscopically poorer, by doing a small amount of damage to the environment. While environmental costs should be taken into account, they don't make wealth a zero-sum game. For example, if you repair a machine that's broken because a part has come unscrewed, you create wealth with no environmental cost.

[5] 普通の車を修理することで、他の皆は微視的には貧しくなるかもしれない。環境への少々のダメージがあるからね。環境のコストは計算に入れなければならないが、そうしたところで富を創ることがゼロサムゲームになるわけじゃない。例えば、ねじが緩んでいただけの機械を直したとしたら、環境への影響なしで富を創り出したことになる。

[5b] This essay was written before Firefox.

[5b] このエッセイは Firefox の出現より前に書かれたものだ。

[6] Many people feel confused and depressed in their early twenties. Life seemed so much more fun in college. Well, of course it was. Don't be fooled by the surface similarities. You've gone from guest to servant. It's possible to have fun in this new world. Among other things, you now get to go behind the doors that say "authorized personnel only." But the change is a shock at first, and all the worse if you're not consciously aware of it.

[6] 20代初めに、混乱して落ち込んでいる人は多い。大学はあんなに面白かったのに。もちろんそうだったろうさ。表面的な類似性にだまされちゃいけない。ゲストから召使いになったわけだからね。この新しい世界で楽しむことだってできる。そのうちのひとつは、「関係者以外立入禁止」のドアの中に入れることだ。でも、この変化は最初は衝撃的だろう。特にその変化を意識しないでいるうちは。

[7] When VCs asked us how long it would take another startup to duplicate our software, we used to reply that they probably wouldn't be able to at all. I think this made us seem naive, or liars.

[7] ベンチャーキャピタルに、別のベンチャー企業が私たちのソフトウェアと同等のものを作るのにどのくらいかかるかと尋ねられたときは、きっと他の人は作れないでしょうと答えていたものだ。ずいぶんものを知らないか、嘘つきだと思われていたに違いない。

[8] Few technologies have one clear inventor. So as a rule, if you know the "inventor" of something (the telephone, the assembly line, the airplane, the light bulb, the transistor) it is because their company made money from it, and the company's PR people worked hard to spread the story. If you don't know who invented something (the automobile, the television, the computer, the jet engine, the laser), it's because other companies made all the money.

[8] 一人の発明者がはっきりしている技術はほとんどない。だから、何か(電話でも、流れ作業でも、飛行機でも、電球でも、トランジスタでも)の「発明者」をあなたが知っているとすれば、それは彼らの会社がそれで金を儲けて、その広告部門が話を広めたからに他ならない。何か(自動車、テレビ、コンピュータ、ジェットエンジン、レーザー)の発明者をあなたが知らないとすれば、それは他の会社がそれで儲けたからだ。

[9] This is a good plan for life in general. If you have two choices, choose the harder. If you're trying to decide whether to go out running or sit home and watch TV, go running. Probably the reason this trick works so well is that when you have two choices and one is harder, the only reason you're even considering the other is laziness. You know in the back of your mind what's the right thing to do, and this

trick merely forces you to acknowledge it.

[9] これは人生においても一般的に良い方法だ。2つの選択肢がある場合、難しいほうを選べ。ジョギングに行くか、座ってテレビを見るか迷ったら、ジョギングに行け。このルールがとてもうまくいくのは、おそらく、選択肢があっても一方が難しいという場合、たぶんあなたの怠惰さがもう一方の選択肢を持ち出したに違いないからだ。心の底では、何をすべきか知っているんだ。このルールに従えば、それを自分で認めることになる。

[10] It is probably no accident that the middle class first appeared in northern Italy and the low countries, where there were no strong central governments. These two regions were the richest of their time and became the twin centers from which Renaissance civilization radiated. If they no longer play that role, it is because other places, like the United States, have been truer to the principles they discovered.

[10] 中産階級が最初に北イタリアと低地諸国に現れたのは偶然ではないだろう。そこには、強い中央政府がなかった。これら 2つの地域は当時最も裕福な地域で、ルネッサンスの拡がりの 2つの中心となった。これらの地域がいまでは中心でないのだとしたら、それは彼らがもともと発見した原理を、米国など他の地域のほうが忠実に実行しているからだろう。

[11] It may indeed be a sufficient condition. But if so, why didn't the Industrial Revolution happen earlier? Two possible (and not incompatible) answers: (a) It did. The Industrial Revolution was one in a series. (b) Because in medieval towns, monopolies and guild regulations initially slowed the development of new means of production.

[11] これは、十分条件でもあり得る。だがそうだとすると、なぜ産業革命はもっと早くに起こらなかったのだろう。2つの(相互に矛盾しない)答えが考えられる。(a)それは実際に起きていた。いわゆる産業革命は、連続して起きていた変化のひとつにすぎない。(b)中世の都市では、独占とギルドの縛りが新しい生産手段の発展を妨げていた。

Chapter 7 MIND THE GAP

第 7 章 格差を考える

May 2004

When people care enough about something to do it well, those who do it best tend to be far better than everyone else. There's a huge gap between Leonardo and second-rate contemporaries like Borgognone. You see the same gap between Raymond Chandler and the average writer of detective novels. A top-ranked professional chess player could play ten thousand games against an ordinary club player without losing once.

何かをより良く成し遂げるということについて人々の関心が集まる場所では、一番の人が、他の人々からずばぬけて良いということが往々にして起こる。レオナルド・ダ・ヴィンチと、同時期のボルゴニョーヌのような 2 番手の画家の間には、とてつもない差がある。同じ格差は、レイモンド・チャンドラーと、平均的な推理小説作家の間にも見られる。トップランクのプロのチェスプレーヤーは、普通のチェスクラブのプレーヤー 1 万人相手にチェスを指して、一度も負けることがないだろう。

Like chess or painting or writing novels, making money is a very specialized skill. But for some reason we treat this skill differently. No one complains when a few people surpass all the rest at playing chess or writing novels, but when a few people make more money than the rest, we get editorials saying this is wrong.

チェスを指したり、絵を描いたり、小説を書くのと同じように、お金を儲けることもまた、非常に特殊な技能である。ところがどういうわけか、私たちはこの技能だけを別のもののように扱う。一握りの人々が残り全てのチェスプレーヤーや小説家をはるかに凌駕することについては誰も文句を言わないのに、一握りの人々が残りの人々より多くの金を儲けると、それは悪いことだという社説が書かれたりする。

Why? The pattern of variation seems no different than for any other skill. What causes people to react so strongly when the skill is making money?

どうしてなんだろう。格差のパターンは他の技能と何ら差がない。それなのに、技能が金を儲けるということになると、どうして人々はこんなに強く反応するのだろうか。

I think there are three reasons we treat making money as different: the misleading model of wealth we learn as children; the disreputable way in which, till recently, most fortunes were accumulated; and the worry that great variations

in income are somehow bad for society. As far as I can tell, the first is mistaken, the second outdated, and the third empirically false. Could it be that, in a modern democracy, variation in income is actually a sign of health?

私たちが金を儲けることを別のものだとするには、次に挙げる 3 つの理由があると思う。子供の時に私たちが学んだ、誤解を招くモデル。近年まで多くの財産が蓄積されてきた、好ましからざる方法。そして収入の格差が大きいことは社会にとって悪いものだという心配。だが私が考えるに、最初のは誤解であり、2 番目のものは時代遅れであり、そして 3 番目は経験的に誤りである。現代の民主主義社会において、収入の格差は実は健全性のしるしであるということはあるまいだろうか。

The Daddy Model of Wealth

富の父親モデル

When I was five I thought electricity was created by electric sockets. I didn't realize there were power plants out there generating it. Likewise, it doesn't occur to most kids that wealth is something that has to be generated. It seems to be something that flows from parents.

5 歳の頃の私は、電気はコンセントで作られているものだと思っていた。電気を生み出している発電所がどこかにあるなんて思いもつかなかったんだ。同じようにたいていの子供は、富が生み出されなければならないものだという事に気付かない。富とは、親から流れ出してくるもののように思っているようだ。

Because of the circumstances in which they encounter it, children tend to misunderstand wealth. They confuse it with money. They think that there is a fixed amount of it. And they think of it as something that's distributed by authorities (and so should be distributed equally), rather than something that has to be created (and might be created unequally).

こういう経験のために、子供は富について誤解しがちだ。富を貨幣と混同してしまうのだ。つまり、世の中にはそれが決まった量しかないと思ってしまう。さらに、それは何らかの権威によって分配されていて(したがって等しく分配されなければならない)、創り出さなければならない(しかも、偏って創り出されるかもしれない)ものとは考えないんだ。

In fact, wealth is not money. Money is just a convenient way of trading one form of wealth for another. Wealth is the underlying stuff—the goods and services we buy. When you travel to a rich or poor country, you don't have to look at people's bank accounts to tell which kind you're in. You can see wealth—in buildings and

streets, in the clothes and the health of the people.

実際には、富は貨幣ではない。貨幣は、富の一形態を別の形態と交換するための、ひとつの便利な方法にすぎない。富は、その下にある、私たちが購入する物やサービスのことだ。富める国と貧しい国に旅行に行ったら、人々の銀行残高をわざわざ調べないでも、どちらの国にいるのかを見て取ることができるだろう。建物や街路、衣服や人々の健康状態なんかにね。

Where does wealth come from? People make it. This was easier to grasp when most people lived on farms, and made many of the things they wanted with their own hands. Then you could see in the house, the herds, and the granary the wealth that each family created. It was obvious then too that the wealth of the world was not a fixed quantity that had to be shared out, like slices of a pie. If you wanted more wealth, you could make it.

これらの富はどこから来るのだろうか？人々が創り出すんだ。ほとんどの人々が農業に従事していて、必要なさまざまなものを自分たちの手で創り出していた時代には、このことはもっと分かりやすかった。家や倉や家畜の群に、それぞれの家族が創り出した富を見て取ることができただろう。世界の富はパイの切れ端のように決まった量を皆で分けるのではないということもまた、明白だったはずだ。もっと富が欲しければ、創ることができた。

This is just as true today, though few of us create wealth directly for ourselves (except for a few vestigial domestic tasks). Mostly we create wealth for other people in exchange for money, which we then trade for the forms of wealth we want. [\[1\]](#)

このことは現在でも真実だが、多少の家庭内の作業を除いては、私たちが自分自身で富を直接創り出すことはほとんどなくなった。多くの人々は、他の人々のために富を創り、それと交換に貨幣を受け取り、次にそれを自分が欲しい富の形態と交換するのだ。 [\[1\]](#)

Because kids are unable to create wealth, whatever they have has to be given to them. And when wealth is something you're given, then of course it seems that it should be distributed equally. [\[2\]](#) As in most families it is. The kids see to that. "Unfair," they cry, when one sibling gets more than another.

子供は富を創り出すことができないから、何であれ所有するものは誰かに与えてもらわなければならない。そして、富が誰かに与えられるものである間は、もちろんそれが等しく分配されなければならないように思える。 [\[2\]](#) 多くの家庭ではそうだろう。子供はちゃんと見ていて、兄弟の誰かがより多くもらうと、「不公平だ」と叫ぶのだ。

In the real world, you can't keep living off your parents. If you want something, you either have to make it, or do something of equivalent value for someone else, in order to get them to give you enough money to buy it. In the real world, wealth is (except for a few specialists like thieves and speculators) something you have to create, not something that's distributed by Daddy. And since the ability and desire to create it vary from person to person, it's not made equally.

実社会では、親にずっと頼って生き続けることはできない。何かが欲しければ、それを自分で作るか、同等の価値あることを他の誰かにしてあげて、その対価を支払ってもらい、それでもって欲しいものを買うしかない。実社会では、富は(泥棒とか山師などの特殊な例を除けば)自分で創り出さねばならないものであって、お父さんに分配してもらうものじゃない。そして、富を創り出す能力と、創りたいと望む欲望は人によって異なるのだから、富は等しくは創り出されない。

You get paid by doing or making something people want, and those who make more money are often simply better at doing what people want. Top actors make a lot more money than B-list actors. The B-list actors might be almost as charismatic, but when people go to the theater and look at the list of movies playing, they want that extra oomph that the big stars have.

人々が欲しがるものを作ったり、欲しがることをしたりすることで、対価が得られる。より多くの対価を受け取るのは、普通は単に人々が欲することをよりうまくできる人だ。トップレベルの役者は、脇役専門の役者よりはるかに多くの金を稼ぐ。脇役専門の役者だってトップスターと同じくらい魅力的であるかもしれないが、人々が映画館に行って上映中の映画のリストを見る時には、ビッグスターの名前を見たがるものなんだ。

Doing what people want is not the only way to get money, of course. You could also rob banks, or solicit bribes, or establish a monopoly. Such tricks account for some variation in wealth, and indeed for some of the biggest individual fortunes, but they are not the root cause of variation in income. The root cause of variation in income, as Occam's Razor implies, is the same as the root cause of variation in every other human skill.

もちろん、人々が欲しがるものを作るだけがお金を得る方法ではない。銀行強盗をしたり、賄賂を要求したり、あるいは独占企業を作るって方法だってある。これらも富の格差の一部ではあるだろうし、実際、最も大きい個人財産のいくつかはそういう方法によったものだろうが、それは富の格差の根本的な理由ではない。富の格差の根本的な理由は、オッカムの剃刀を適用すれば、人それぞれの技能の格差の根本にあるものと同じだ。

In the United States, the CEO of a large public company makes about 100 times as much as the average person. [3] Basketball players make about 128 times as much, and baseball players 72 times as much. Editorials quote this kind of statistic with horror. But I have no trouble imagining that one person could be 100 times as productive as another. In ancient Rome the price of *slaves* varied by a factor of 50 depending on their skills. [4] And that's without considering motivation, or the extra leverage in productivity that you can get from modern technology.

米国では大きな公開企業の CEO は、平均的な人の 100 倍くらいの給料を受け取る。[3] バスケットボール選手は 128 倍、野球選手は 72 倍くらいだ。新聞の社説ではこのような統計が、恐れをもって語られる。でも、ある人が別の人より 100 倍生産的である、ということ想像するのはさして難しくない。古代ローマでは、奴隷の価格はその技能によって 50 倍ほども差があったという。[4] しかもそれには、動機づけや、現代の技術から得られる生産性の増幅効果は入っていないわけだ。

Editorials about athletes' or CEOs' salaries remind me of early Christian writers, arguing from first principles about whether the Earth was round, when they could just walk outside and check. [5] How much someone's work is worth is not a policy question. It's something the market already determines.

スポーツ選手や CEO の給料について語る社説は、地球が丸いかどうかという第一原理を延々と議論していた昔のキリスト教徒たちを思い起こさせる。ちょっと外に出て調べれば分かることなのに。[5] 誰かの仕事がどのくらいの価値があるのかは、政府の方針の問題ではない。それは既に市場が決めていることだ。

"Are they really worth 100 of us?" editorialists ask. Depends on what you mean by worth. If you mean worth in the sense of what people will pay for their skills, the answer is yes, apparently.

「彼らは本当に私たちの 100 倍の価値があるのだろうか」と社説は問う。この問いの意味は、価値という言葉で何を意味するかによる。もし価値というのを、人々が彼らの技能に支払うものという意味でとれば、答えは明らかに、イエスだ。

A few CEOs' incomes reflect some kind of wrongdoing. But are there not others whose incomes really do reflect the wealth they generate? Steve Jobs saved a company that was in a terminal decline. And not merely in the way a turnaround specialist does, by cutting costs; he had to decide what Apple's next products should be. Few others could have done it. And regardless of the case with CEOs,

it's hard to see how anyone could argue that the salaries of professional basketball players don't reflect supply and demand.

何人かの CEO の収入は不正な行いによってもたらされているかもしれない。だが、実際に生み出した富の量を反映した報酬を受け取っている CEO がいないなんてことはなかろう。スティーブ・ジョブズは、末期的な凋落状態にある会社を救った。しかも会社再建の専門家がよくやるように、経費を削減するだけで救ったわけじゃない。彼はアップルが次に何を作るべきかを決めたんだ。それができる人が他にいたろうか。それに、CEO の場合を置いといても、プロのバスケットボール選手の給料が需要と供給を反映しているということを認めない人はいないだろう。

It may seem unlikely in principle that one individual could really generate so much more wealth than another. The key to this mystery is to revisit that question, are they really worth 100 of us? *Would* a basketball team trade one of their players for 100 random people? What would Apple's next product look like if you replaced Steve Jobs with a committee of 100 random people? [6] These things don't scale linearly. Perhaps the CEO or the professional athlete has only ten times (whatever that means) the skill and determination of an ordinary person. But it makes all the difference that it's concentrated in one individual.

一人の個人が他人よりそんなに多くの富を生み出せることが原理的にあり得るとは信じ難いかもしれない。このミステリーの鍵は、最初の問いに戻ってみることだ。彼は、本当に私たちの 100 倍の価値があるのだろうか。バスケットボールチームは、彼らの選手の一人を、適当に選んだ 100 人の人々と交換したいと思うだろうか。スティーブ・ジョブズを、適当に選んだ 100 人の人々からなる委員会と置き換えたら、アップルの次の製品はどうなるだろうか。[6] こういうものごとは、線形には拡大しないんだ。CEO やプロのスポーツ選手は、普通の人々の、たかだか 10 倍くらいの技能や覚悟(その意味するところが何であれ)を持っているだけかもしれない。でも、それが一人の人間の中に凝縮されているということが、大きな違いとなるのだ。

When we say that one kind of work is overpaid and another underpaid, what are we really saying? In a free market, prices are determined by what buyers want. People like baseball more than poetry, so baseball players make more than poets. To say that a certain kind of work is underpaid is thus identical with saying that people want the wrong things.

ある種の仕事は給料を貰い過ぎていたりとか、給料が少な過ぎるとか言うとき、実際には何を見ているのだろうか。自由市場では、価格は書き手が欲しいと思うことで決定される。人々は詩よりもバスケットボールが好きだから、詩人よりバスケットボール選手が多くの

給料を受け取る。だから、ある仕事に対する給料が少な過ぎるというのは、人々が間違っただけのものを望んでいると言っているのと同じだ。

Well, of course people want the wrong things. It seems odd to be surprised by that. And it seems even odder to say that it's *unjust* that certain kinds of work are underpaid. [7] Then you're saying that it's unjust that people want the wrong things. It's lamentable that people prefer reality TV and corndogs to Shakespeare and steamed vegetables, but unjust? That seems like saying that blue is heavy, or that up is circular.

もちろん、人々は間違っただけのものを望むものだ。そんなことに驚くほうがおかしい。ましてや、ある種の仕事の給料が少な過ぎるのは不公正だなんて言うのはもっとおかしい。[7] 人々が間違っただけのものを望むのは不公正だと言っているのと同じだからだ。そりゃあ、人々がシェークスピアよりバラエティ番組を、温野菜よりアメリカンドッグを好むことは、残念ではある。だが、不公正だって?それはまるで、青色は重いとか、上は丸いとか言うようなものだ。

The appearance of the word "unjust" here is the unmistakable spectral signature of the Daddy Model. Why else would this idea occur in this odd context? Whereas if the speaker were still operating on the Daddy Model, and saw wealth as something that flowed from a common source and had to be shared out, rather than something generated by doing what other people wanted, this is exactly what you'd get on noticing that some people made much more than others.

ここで「不公正」という言葉が出てくるということは、父親モデルが心に刻まれている間違いない証拠だ。でなければ、どうしてこんなとんちんかんな考えが出てくるんだい?未だに父親モデルを信じていて、富は人が欲することをやることで生み出されるものではなく、共有の源から流れてきて分配されるものだと思っているからこそ、誰かが他の人よりずっとたくさん儲けているのを見て不公正だと思ってしまうんだ。

When we talk about "unequal distribution of income," we should also ask, where does that income come from? [8] Who made the wealth it represents? Because to the extent that income varies simply according to how much wealth people create, the distribution may be unequal, but it's hardly unjust.

「収入の不均一な分布」に関して話す時には、その収入がどこから来たのかについても考えなくちゃならない。その収入の元になった富は誰が作ったんだろうか。[8] 収入の多寡が人が創り出した富の多寡による限り、その分布は不均一になるだろうが、不公正とはいえない。

Stealing It

盗むこと

The second reason we tend to find great disparities of wealth alarming is that for most of human history the usual way to accumulate a fortune was to steal it: in pastoral societies by cattle raiding; in agricultural societies by appropriating others' estates in times of war, and taxing them in times of peace.

富の大きな偏りに対して私たちが警戒する 2 番目の理由は、人類の歴史の大部分において、財産の蓄積は盗むことによって行われるのが普通だったからだ。牧畜社会では畜牛の強奪によって。農業社会では戦争時に他人の土地を奪い、平時には税をかけることによって。

In conflicts, those on the winning side would receive the estates confiscated from the losers. In England in the 1060s, when William the Conqueror distributed the estates of the defeated Anglo-Saxon nobles to his followers, the conflict was military. By the 1530s, when Henry VIII distributed the estates of the monasteries to his followers, it was mostly political. [\[9\]](#) But the principle was the same. Indeed, the same principle is at work now in Zimbabwe.

衝突が起きると、勝者は敗者の土地を没収する。英国で 1060 年代に、ウィリアム 1 世が敗者であるアングロサクソン貴族の土地を自分の追従者に分配したときは、衝突は軍事的なものだった。1530 年代までに、ヘンリー 8 世が僧院の土地を自分の臣下に分配したときは、衝突は政治的なものだった。[\[9\]](#) だが、原則は同じだ。実際、同じ原則が現在のジンバブエで進行している。

In more organized societies, like China, the ruler and his officials used taxation instead of confiscation. But here too we see the same principle: the way to get rich was not to create wealth, but to serve a ruler powerful enough to appropriate it. 中国のようにより構造化された社会では、支配者と役人は没収する代わりに税を使った。でも、ここでも同じ原理を見ることができる。財をなすには、富を創り出すのではなく、搾取できるだけの力を持つ支配者に仕えることだ。

This started to change in Europe with the rise of the middle class. Now we think of the middle class as people who are neither rich nor poor, but originally they were a distinct group. In a feudal society, there are just two classes: a warrior aristocracy, and the serfs who work their estates. The middle class were a new, third group who lived in towns and supported themselves by manufacturing and

trade.

ヨーロッパで中産階級が台頭してきたことから、この原理は変わり始めた。現在では、中産階級というと裕福でも貧乏でもない人々を指すと考えられているが、本来それは別の集団だったのだ。封建社会には 2 つの階級しかなかった。諸侯・騎士と、彼らの土地で働く農奴である。中産階級は新しい 3 番目の集団であり、都市に住んで製造業と商業で生活を為した人々であった。

Starting in the tenth and eleventh centuries, petty nobles and former serfs banded together in towns that gradually became powerful enough to ignore the local feudal lords. [10] Like serfs, the middle class made a living largely by creating wealth. (In port cities like Genoa and Pisa, they also engaged in piracy.) But unlike serfs they had an incentive to create a lot of it. Any wealth a serf created belonged to his master. There was not much point in making more than you could hide. Whereas the independence of the townsmen allowed them to keep whatever wealth they created.

10 世紀から 11 世紀にかけて、弱小貴族とかつて農奴であった人々が都市に集まり、次第にその地の領主に従わないだけの力を持ち始めた。[10] 農奴と同じように、中産階級は主として富を創り出すことを生活の糧としていた(ジェノバやピサのような湾岸都市では海賊稼業も行われたが)。だが農奴と違い、中産階級には富をたくさん創り出す動機があった。農奴はいくら富を創り出しても、領主のものになってしまう。それなら自分で隠しもっておける以上の富を創る意味はあまりない。しかし独立した都市住民は、創れば創るだけ自分で持っておくことができた。

Once it became possible to get rich by creating wealth, society as a whole started to get richer very rapidly. Nearly everything we have was created by the middle class. Indeed, the other two classes have effectively disappeared in industrial societies, and their names been given to either end of the middle class. (In the original sense of the word, Bill Gates is middle class.)

富を創り出すことで裕福になることが可能になった時から、社会は全体として、急速に裕福になっていった。現在私たちが持っているほとんどのものは、中産階級が創り出したものだ。実際、工業社会においては他の 2 つの階級はほとんど消滅してしまい、その名前だけが中産階級の両端の人々に対して使われているにすぎない(言葉の本来の意味から言えば、ビル・ゲイツも中産階級だ。)

But it was not till the Industrial Revolution that wealth creation definitively replaced corruption as the best way to get rich. In England, at least, corruption

only became unfashionable (and in fact only started to be called "corruption") when there started to be other, faster ways to get rich.

そして産業革命が、裕福になる最良の方法を、汚職から富の創出へと決定的に置き換えた。少なくとも英国では、他の、より速く裕福になる方法が現れた後では、汚職は流行らなくなった(そして、その時になってそれが「汚職」と呼ばれるようになったのだ)。

Seventeenth-century England was much like the third world today, in that government office was a recognized route to wealth. The great fortunes of that time still derived more from what we would now call corruption than from commerce. [11] By the nineteenth century that had changed. There continued to be bribes, as there still are everywhere, but politics had by then been left to men who were driven more by vanity than greed. Technology had made it possible to create wealth faster than you could steal it. The prototypical rich man of the nineteenth century was not a courtier but an industrialist.

17世紀の英国は、役所に入ることが富への道と考えられていたという点で、現代の第三世界とよく似ている。当時の多くの財産は、商業よりも、私たちが現在では汚職と呼んでいる行為によってもたらされた。[11] 19世紀までにそれは変わった。賄賂は常にあったし今でもなくなっていないが、その時点までには、政治は欲望よりもむしろ虚栄心を満たす場となっていた。技術が、盗むよりも速く富を創り出すことを可能にしたのだ。19世紀の典型的な資産家は、廷臣ではなく産業資本家であった。

With the rise of the middle class, wealth stopped being a zero-sum game. Jobs and Wozniak didn't have to make us poor to make themselves rich. Quite the opposite: they created things that made our lives materially richer. They had to, or we wouldn't have paid for them.

中産階級の台頭により、富はゼロサムゲームではなくなった。ジョブズとウォズニアックは、自分が裕福になるために残りの人々を貧しくする必要はなかった。むしろ逆なんだ。彼らは、私たちの生活を物質的により豊かにするものを創り出したんだ。彼らがそうしなかったら、私たちは彼らにお金を支払いはしなかっただろう。

But since for most of the world's history the main route to wealth was to steal it, we tend to be suspicious of rich people. Idealistic undergraduates find their unconsciously preserved child's model of wealth confirmed by eminent writers of the past. It is a case of the mistaken meeting the outdated.

しかし、世界の歴史のほとんどの部分で、富への道が盗むことであったという事実から、私たちは裕福な人々について疑いの目を向けてしまう。理想に満ちた学生が過去の著名な作

家の本を読んでいて、自分が無意識に信じていた、子供の富のモデルを承認する記述を目にすることがある。「間違い」と「時代遅れ」の不幸な出会いだ。

"Behind every great fortune, there is a crime," Balzac wrote. Except he didn't. What he actually said was that a great fortune with no apparent cause was probably due to a crime well enough executed that it had been forgotten. If we were talking about Europe in 1000, or most of the third world today, the standard misquotation would be spot on. But Balzac lived in nineteenth-century France, where the Industrial Revolution was well advanced. He knew you could make a fortune without stealing it. After all, he did himself, as a popular novelist. [\[12\]](#)

「全ての大きな財産の影には、犯罪がある」とバルザックは書いた、とされているが、実は彼はそんなことは書いていない。彼が実際に言ったのはこうだ。明白な理由のない大きな財産はおそらく犯罪によるもので、それがあまりに巧妙だったために忘れ去られてしまったのだろう、と。西暦 1000 年のヨーロッパだとか、現代の第三世界の多くでは、この引用間違いはそれでも該当することがあるだろう。だがバルザックは産業革命が十分に進んだ 19 世紀のフランスにいたのだ。彼は、盗まなくても財産をなすことができると知っていた。結局、彼自身も人気作家としてそうしたわけだ。 [\[12\]](#)

Only a few countries (by no coincidence, the richest ones) have reached this stage. In most, corruption still has the upper hand. In most, the fastest way to get wealth is by stealing it. And so when we see increasing differences in income in a rich country, there is a tendency to worry that it's sliding back toward becoming another Venezuela. I think the opposite is happening. I think you're seeing a country a full step ahead of Venezuela.

ほんのいくつかの国(必然的に最も富める国々)だけがこの段階に達している。残りの大部分では、汚職が依然として上手を取っている。そこでは富を得る最も手っ取り早い方法は盗むことだ。だから、富める国で収入の格差が開きつつあるのを見ると、私たちはもうひとつのベネズエラへと落ちつつあるんじゃないか、と心配してしまうのだ。でも実際は全く逆だと思う。その国は、ベネズエラから一歩先を進んでいるんだ。

The Lever of Technology

技術の梃子

Will technology increase the gap between rich and poor? It will certainly increase the gap between the productive and the unproductive. That's the whole point of technology. With a tractor an energetic farmer could plow six times as much land

in a day as he could with a team of horses. But only if he mastered a new kind of farming.

技術は富める者と貧しい者の格差を拡大するだろうか。技術は確かに、生産的な者とそうでない者との格差を拡大する。それが技術というものだからだ。耕耘機があれば、精力的な農家は、一群の馬で一日に耕せる土地の 6 倍もの土地を耕せるだろう。だが、それは彼が新しい農法を身に付けたら、の話だ。

I've seen the lever of technology grow visibly in my own time. In high school I made money by mowing lawns and scooping ice cream at Baskin-Robbins. This was the only kind of work available at the time. Now high school kids could write software or design web sites. But only some of them will; the rest will still be scooping ice cream.

私自身、技術の梃子が伸びていくのを自分の目で目撃した。高校の時、私は芝刈りとサーティーワン・アイスクリームの売り子をして小遣いを得ていた。当時はそれぐらいしかできることがなかったんだ。今の高校生は、ソフトウェアを書いたり Web サイトをデザインしたりすることができる。でも、一部の人だけだ。残りはやっぱりアイスクリームを売っているだろう。

I remember very vividly when in 1985 improved technology made it possible for me to buy a computer of my own. Within months I was using it to make money as a freelance programmer. A few years before, I couldn't have done this. A few years before, there was no such *thing* as a freelance programmer. But Apple created wealth, in the form of powerful, inexpensive computers, and programmers immediately set to work using it to create more.

1985 年に、技術の進歩によって自分のコンピュータを買うことができるようになった時のことを、私は鮮明に覚えている。数ヶ月のうちに、私はそれを使ってフリーランスプログラマとして金を稼いでいた。その数年前までは、そんなことはやりたくてもできなかったんだ。その数年前には「フリーランスプログラマ」なんていうモノ自体が存在しなかった。でもアップルが強力で安価なコンピュータという形で富を創り出し、プログラマは直ちにそれを使ってより多くの富を創り始めたんだ。

As this example suggests, the rate at which technology increases our productive capacity is probably polynomial, rather than linear. So we should expect to see ever-increasing variation in individual productivity as time goes on. Will that increase the gap between rich and the poor? Depends which gap you mean.

この例が示すように、技術が私たちの生産性を増加させる率は、線形ではなく多項式であ

るだろう。だから時が経つにつれ、個人の生産性の差は増加し続けると思わなくちゃならない。それは、富める者と貧しい者の格差を拡大してゆくだろうか。「格差」が何を意味するかによる。

Technology should increase the gap in income, but it seems to decrease other gaps. A hundred years ago, the rich led a different *kind* of life from ordinary people. They lived in houses full of servants, wore elaborately uncomfortable clothes, and travelled about in carriages drawn by teams of horses which themselves required their own houses and servants. Now, thanks to technology, the rich live more like the average person.

技術は確かに収入の格差を広げるだろうが、他の格差を縮めるように思われる。100 年前は、裕福な人々は普通の人々とは別種の生活をしていた。召使いをたくさん抱えた屋敷に住み、不自然に着にくそうな衣服をまとい、何頭もの馬に牽かれた馬車で出かけた。その馬車だって、専用の厩舎と御者が必要だった。現代では技術のおかげで、裕福な人々の生活は普通の人々の生活と似たようなものになっている。

Cars are a good example of why. It's possible to buy expensive, handmade cars that cost hundreds of thousands of dollars. But there is not much point. Companies make more money by building a large number of ordinary cars than a small number of expensive ones. So a company making a mass-produced car can afford to spend a lot more on its design. If you buy a custom-made car, something will always be breaking. The only point of buying one now is to advertise that you can.

車が、その理由の良い例だ。高価な手作りの車を何千万も出して買うことはできるかもしれない。でもそれをする意味はあまりない。企業は、少数の高価な車を作るより、普通の車を大量生産するほうが儲けることができる。したがって、大量生産する車を作っている企業は設計により多くの金をかけることができる。カスタムメイドの車を買ったとしたら、きっといつもどこかが壊れていることだろう。現代においてそういう車を買う意味は、あなたがそれを買えるんだということを宣伝することくらいしかない。

Or consider watches. Fifty years ago, by spending a lot of money on a watch you could get better performance. When watches had mechanical movements, expensive watches kept better time. Not any more. Since the invention of the quartz movement, an ordinary Timex is more accurate than a Patek Philippe costing hundreds of thousands of dollars. [\[13\]](#) Indeed, as with expensive cars, if you're determined to spend a lot of money on a watch, you have to put up with

some inconvenience to do it: as well as keeping worse time, mechanical watches have to be wound.

あるいは腕時計を考えてみるとよい。50年前なら、金をたくさん出せば良い性能の腕時計が買えた。機械式腕時計の時代には、高価な腕時計はより正確に時を刻んだ。もはや時代はそうではない。クォーツ式時計の発明以来、普通の TIMEX は何千万もするパテックフィリップよりも正確になった。[\[13\]](#) 実際、高価な車と同様に、腕時計にたくさん金をかけたなら、若干の不便を我慢しなくちゃならない。機械式時計は不正確な上に、ねじを巻かなくちゃならないからね。

The only thing technology can't cheapen is brand. Which is precisely why we hear ever more about it. Brand is the residue left as the substantive differences between rich and poor evaporate. But what label you have on your stuff is a much smaller matter than having it versus not having it. In 1900, if you kept a carriage, no one asked what year or brand it was. If you had one, you were rich. And if you weren't rich, you took the omnibus or walked. Now even the poorest Americans drive cars, and it is only because we're so well trained by advertising that we can even recognize the especially expensive ones. [\[14\]](#)

ブランドは唯一、技術で安価にできないものだ。現在、ブランドのことをしょっちゅう耳にするようになったのはそのせいだ。裕福な者と貧しい者の物質的な差が消滅して、ブランドだけが残されているんだ。でも自分の物にどんなラベルが付いているかなんて、それを持つことと持たないこととの差に比べれば取るに足らないことじゃないか。1900年には、自分で馬車を持っていたら、誰もそれが何年式でどこのブランドかなんて尋ねなかったろう。それを持ってさえいれば裕福だったんだ。裕福でなければ、乗り合い馬車に乗るか、てくてく歩くしかなかった。現代では、最も貧しい米国人でも車に乗っている。広告をさんざん見せられたせいで、特に高価な車を見分けることができるから、差を感じているだけなんだ。[\[14\]](#)

The same pattern has played out in industry after industry. If there is enough demand for something, technology will make it cheap enough to sell in large volumes, and the mass-produced versions will be, if not better, at least more convenient. [\[15\]](#) And there is nothing the rich like more than convenience. The rich people I know drive the same cars, wear the same clothes, have the same kind of furniture, and eat the same foods as my other friends. Their houses are in different neighborhoods, or if in the same neighborhood are different sizes, but within them life is similar. The houses are made using the same construction techniques and contain much the same objects. It's inconvenient to do something

expensive and custom.

同じパターンがさまざまな産業分野で繰り返されてきた。何かについて十分な需要があれば、それは技術の発展によって大量に売りさばけるレベルまで安くなり、大量生産されたものは、より良くはないにせよ、少なくともより便利になる。[\[15\]](#) そして、裕福な人々は何より便利さを好む。私が知っている金持ちの人々は、私の他の友人たちと同じ車に乗り、同じ服を着て、同じような家具を持ち、そして同じ食事をしている。彼らの家は高級住宅地にあり、そうでなくても周りの家と大きさが違うが、その中の生活は似たようなものだ。家だって、同じ製造技術を使って、似たようなもので作られている。そうでない、高価なあつらえものをしようと思えば、不便になるだけだ。

The rich spend their time more like everyone else too. Bertie Wooster seems long gone. Now, most people who are rich enough not to work do anyway. It's not just social pressure that makes them; idleness is lonely and demoralizing.

それに金持ちの時間の使い方だって、他のみんなと似たようなものだ。ウッドハウスが書くジーヴスとパーティのパーティ・ウースターのような生活は、もう過去のもんだ。現代では、働かないでもよいくらい裕福な人の多くは、やっぱり働いている。社会的にそうしないと恰好がつかないというだけでなく、何もしないでいるのは寂しいし、つまらないからだ。

Nor do we have the social distinctions there were a hundred years ago. The novels and etiquette manuals of that period read now like descriptions of some strange tribal society. "With respect to the continuance of friendships..." hints *Mrs. Beeton's Book of Household Management* (1880), "it may be found necessary, in some cases, for a mistress to relinquish, on assuming the responsibility of a household, many of those commenced in the earlier part of her life." A woman who married a rich man was expected to drop friends who didn't. You'd seem a barbarian if you behaved that way today. You'd also have a very boring life. People still tend to segregate themselves somewhat, but much more on the basis of education than wealth. [\[16\]](#)

また、100 年前にあった社会的な区別ももはや存在しない。当時の小説やエチケットマニュアルを読むと、まるで何か奇妙な種族の社会を覗いているように感じられる。『ビートン夫人の賢い主婦マニュアル』（1880 年刊行）によれば、「友人関係の継続に関してですが、家庭を守る責任を考えると、夫人は以前の生活で得ていた友人関係の多くを手放すことが必要になる場合もあります」とある。裕福な夫と結婚した女性は、そうでない友人と関係を絶つことが期待されていたんだ。現在そんなことをしたらまるで別世界から来た人のように思われるだろう。それにそういう人生はとても退屈になりそうだ。人々はまだ確かに

多少は階層によって分かれた生活を送るが、それは富よりもむしろ教育に基づく理由が大きいだろう。[\[16\]](#)

Materially and socially, technology seems to be decreasing the gap between the rich and the poor, not increasing it. If Lenin walked around the offices of a company like Yahoo or Intel or Cisco, he'd think communism had won. Everyone would be wearing the same clothes, have the same kind of office (or rather, cubicle) with the same furnishings, and address one another by their first names instead of by honorifics. Everything would seem exactly as he'd predicted, until he looked at their bank accounts. Oops.

物質的にも社会的にも、技術は富める者と貧しい者の格差を拡げるのではなく縮めている。レーニンが Yahoo や Intel や Cisco のオフィスを見学したら、社会主義は勝ったと考えただろうね。だって全員が同じ服を着て、同じような調度を備えたオフィス(というよりブース)で仕事し、お互いを敬称ではなく名前で呼び合っている。全てがまさに彼が予想した通りではないか。ただ、銀行の口座を見たらびっくりするだろうけどね。

Is it a problem if technology increases that gap? It doesn't seem to be so far. As it increases the gap in income, it seems to decrease most other gaps.

技術が格差を拡大するとしたら、それは問題だろうか。今のところ、そのようには思えない。収入の格差が広がっても他の多くの格差が縮まっているからだ。

Alternative to an Axiom

別の公理

One often hears a policy criticized on the grounds that it would increase the income gap between rich and poor. As if it were an axiom that this would be bad. It might be true that increased variation in income would be bad, but I don't see how we can say it's *axiomatic*.

富裕層と貧困層の収入格差を拡げるという理由から政策が批判されるのを耳にしたことがあるだろう。まるで、そのこと自体が悪いことだという公理があるみたいに。収入格差の拡大が何らかの悪影響を持つということはあるかもしれないが、なんでそれが自明の公理のように扱われなくちゃならないのか、私には分からない。

Indeed, it may even be false, in industrial democracies. In a society of serfs and warlords, certainly, variation in income is a sign of an underlying problem. But serfdom is not the only cause of variation in income. A 747 pilot doesn't make 40

times as much as a checkout clerk because he is a warlord who somehow holds her in thrall. His skills are simply much more valuable.

実際、それは工業化された民主社会では真実ではないかもしれないじゃないか。農奴と領主の社会では、確かに収入の格差は何かの問題のしるしであった。だが、農奴制は収入の格差の唯一の問題ではない。747 ジャンボジェットのパイロットは、荷物預り窓口スタッフの 40 倍の収入を得ているかもしれないが、だからといってパイロットが領主で他のスタッフを奴隷にしているなんてわけじゃない。単に彼の技能のほうがずっと価値があるからだ。

I'd like to propose an alternative idea: that in a modern society, increasing variation in income is a sign of health. Technology seems to increase the variation in productivity at faster than linear rates. If we don't see corresponding variation in income, there are three possible explanations: (a) that technical innovation has stopped, (b) that the people who would create the most wealth aren't doing it, or (c) that they aren't getting paid for it.

ここで、別の考え方を提唱してみたい。現代社会では、収入の格差はむしろ健全性のしるしであるというものだ。技術は生産性の格差を線形以上の比率で拡大する。もし収入がそれと同じ比率で拡大しないのなら、そこには 3 つばかりの説明が考えられる。(a) 技術革新が止まってしまったか、(b) 富を創れる人々が富を創っていないか、(c) 富を創り出している人々がその富に対する対価を得ていないか、だ。

I think we can safely say that (a) and (b) would be bad. If you disagree, try living for a year using only the resources available to the average Frankish nobleman in 800, and report back to us. (I'll be generous and not send you back to the stone age.)

(a)と(b)の理由については、悪いことだと言ってしまってよいだろう。反対するなら、800 年のフランク王国の貴族と同じ資源だけを使って 1 年間生活してみてから、どうだったかを教えてほしい(石器時代と言わなかっただけましだろう?)。

The only option, if you're going to have an increasingly prosperous society without increasing variation in income, seems to be (c), that people will create a lot of wealth without being paid for it. That Jobs and Wozniak, for example, will cheerfully work 20-hour days to produce the Apple computer for a society that allows them, after taxes, to keep just enough of their income to match what they would have made working 9 to 5 at a big company.

したがって、ますます繁栄しているにもかかわらず収入の格差が増加しない社会があった

ら、考えられる可能性は(c)しかない。すなわち、人々は富をたくさん創り出し、しかしその対価を受け取らないという社会だ。例えば、ジョブズとウォズニアックが一日 20 時間を費やして Apple コンピュータを作り出し、でも税引き後の給料が大企業で 9 時から 5 時まで働いている人と同じくらいであるような社会だ。

Will people create wealth if they can't get paid for it? Only if it's fun. People will write operating systems for free. But they won't install them, or take support calls, or train customers to use them. And at least 90% of the work that even the highest tech companies do is of this second, unedifying kind.

対価を得ないでも人々は富を創り出すだろうか。それ自体が楽しいことなら、そうするだろう。ただでオペレーティングシステムを書いてしまう人だっている。でも彼らだって、それをインストールし、サポートの電話を受け、顧客がそれを使えるように教えるというところまではやらない。どんなハイテク企業だって、少なくとも 90%の仕事は、後者のあまり面白くない種類のものだ。

All the unfun kinds of wealth creation slow dramatically in a society that confiscates private fortunes. We can confirm this empirically. Suppose you hear a strange noise that you think may be due to a nearby fan. You turn the fan off, and the noise stops. You turn the fan back on, and the noise starts again. Off, quiet. On, noise. In the absence of other information, it would seem the noise is caused by the fan.

このようなやっても楽しくない種類の富の創造は、社会が私有財産を没収するようになると劇的に減少する。私たちは経験的にそれを確認することができる。例えば妙な雑音が聞こえてきて、それが近くの扇風機によるもののように思えたとしよう。扇風機を止めると、音は止まる。扇風機を動かすと、また音が聞こえる。オフにすると静寂。オンにすると雑音。他の情報がなければ、雑音は扇風機が原因だと考えられる。

At various times and places in history, whether you could accumulate a fortune by creating wealth has been turned on and off. Northern Italy in 800, off (warlords would steal it). Northern Italy in 1100, on. Central France in 1100, off (still feudal). England in 1800, on. England in 1974, off (98% tax on investment income). United States in 1974, on. We've even had a twin study: West Germany, on; East Germany, off. In every case, the creation of wealth seems to appear and disappear like the noise of a fan as you switch on and off the prospect of keeping it.

歴史上のさまざまな場所と時代で、富を創り出した人が財産を蓄えられるかどうかはオン

になったりオフになったりしてきた。800年、北イタリア、オフ(領主による搾取)。1100年、北イタリア、オン。1100年、中央フランス、オフ(まだ封建制)。1800年、イギリス、オン。1974年、イギリス、オフ(投資による収入に対する98%の課税)。1974年、米国、オン。双子による調査だってある。西ドイツ、オン。東ドイツ、オフ。どの場合でも、富を蓄えられるかどうかによって、富の創造は現れたり消えたりしてきた。ちょうど扇風機のスイッチで雑音が聞こえたり消えたりするみたいに。

There is some momentum involved. It probably takes at least a generation to turn people into East Germans (luckily for England). But if it were merely a fan we were studying, without all the extra baggage that comes from the controversial topic of wealth, no one would have any doubt that the fan was causing the noise.

もちろん、若干の慣性はある。人々が東ドイツ人になるまでには(英国にとっては幸いなことに)おそらく一世代を必要としたろう。でも、富に関するさまざまな議論を呼ぶ言説を置いておいて、扇風機の原理に注目すれば、扇風機が雑音の理由になっていることを疑う人はいないだろう。

If you suppress variations in income, whether by stealing private fortunes, as feudal rulers used to do, or by taxing them away, as some modern governments have done, the result always seems to be the same. Society as a whole ends up poorer.

封建制の領主がやったように私有財産を盗むにせよ、いくつかの近代の政府がやったように税金でそれを奪うにせよ、収入の格差を抑えれば、結果はいつも同じだ。社会は全体として貧しくなる。

If I had a choice of living in a society where I was materially much better off than I am now, but was among the poorest, or in one where I was the richest, but much worse off than I am now, I'd take the first option. If I had children, it would arguably be immoral not to. It's absolute poverty you want to avoid, not relative poverty. If, as the evidence so far implies, you have to have one or the other in your society, take relative poverty.

もし、現在よりも物質的にずっと豊かで、でも社会的には最下層であるような状況と、社会的に一番の富裕層であるが、現在よりも貧しい状況を選べと言われたら、私は前者を選ぶ。私に子供がいたとしたら、そちらを選ばないことは倫理に反することにさえるだろう。避けたいのは絶対的な貧困であって、相対的な貧困じゃない。ここまで挙げた証拠を考えれば、もしあなたの社会の中で選択の余地があるなら、あなたは相対的な貧困を選ぶはずだ。

You need rich people in your society not so much because in spending their money they create jobs, but because of what they have to do to *get* rich. I'm not talking about the trickle-down effect here. I'm not saying that if you let Henry Ford get rich, he'll hire you as a waiter at his next party. I'm saying that he'll make you a tractor to replace your horse.

社会に裕福な人々が必要なのは、別に彼らが金を使うことで雇用が生まれるからじゃない。彼らが裕福になるに当たってなしたことのためであるんだ。トリクルダウン効果のことを言っているんじゃないよ。ヘンリー・フォードが金持ちになれば、彼はあなたを次のパーティでの給仕として雇ってくれるだろう、ということではない。重要なのは、彼が、あなたに耕耘機を作ってくれて、それがあなたの馬を置き換える、ということなんだ。

Notes

注釈

[1] Part of the reason this subject is so contentious is that some of those most vocal on the subject of wealth—university students, heirs, professors, politicians, and journalists—have the least experience creating it. (This phenomenon will be familiar to anyone who has overheard conversations about sports in a bar.)

[1] この話題に多くの異論がある理由のひとつは、一番富を話題にしたがる人々(大学の学生、資産家の二代目、教授、政治家、ジャーナリスト等)が、富が創り出すことに関して一番経験を持たない人々だからだ(この現象は、酒場でのスポーツ談義を耳にしたことのある人には覚えがあるだろう)。

Students are mostly still on the parental dole, and have not stopped to think about where that money comes from. Heirs will be on the parental dole for life. Professors and politicians live within socialist eddies of the economy, at one remove from the creation of wealth, and are paid a flat rate regardless of how hard they work. And journalists as part of their professional code segregate themselves from the revenue-collecting half of the businesses they work for (the ad sales department). Many of these people never come face to face with the fact that the money they receive represents wealth—wealth that, except in the case of journalists, someone else created earlier. They live in a world in which income is doled out by a central authority according to some abstract notion of fairness (or randomly, in the case of heirs), rather than given by other people in return for something they wanted, so it may seem to them unfair that things don't work the

same in the rest of the economy.

学生はたいてい、まだ親の扶養下にあり、その金がどこから来たかについて立ち止まって考えたことはない。資産家の二代目は一生親の庇護の下にある。教授と政治家は直接富を創る現場から一步離れており、どれだけ頑張っても給料は変わらないという、経済に生じる社会主義の渦の中に生きている。ジャーナリストは、その憲章によって、自分の会社の利益を集める部門(広告宣伝部)からは離れたところにいる。これらの人々の多くは、自分が受け取るお金が、(ジャーナリストの場合を除いては)誰か他の人が作り出した富を表現しているということに直面する経験をしない。彼らは実際に、中央の権威が、何か抽象的な公平さ(あるいは、資産家の二代目の場合はランダムな要素)によって給料を分配しているという世界に生きているんだ。人が欲しがるものを作ることで給料を受け取るのになしにね。だから、経済の他の部分が同じようになってないのを見て、不公平を感じるんだ。

(Some professors do create a great deal of wealth for society. But the money they're paid isn't a *quid pro quo*. It's more in the nature of an investment.)

(社会に非常に大きな富をもたらしている教授も確かにいる。だが、そういう教授がもらう給料は交換原則(*quid pro quo*)にはいない。むしろその本来的な性格は投資に近い。)

[2] When one reads about the origins of the Fabian Society, it sounds like something cooked up by the high-minded Edwardian child-heroes of Edith Nesbit's *The Wouldbegoods*.

[2] ファビアン協会の起源を読むと、まるでエディス・ネスビットの『よい子同盟』の主人公の、エドワード朝時代の子供たちが考え出したもののように思える。

[3] According to a study by The Corporate Library, the median total compensation, including salary, bonus, stock grants, and the exercise of stock options, of S&P 500 CEOs in 2002 was \$3.65 million. According to *Sports Illustrated*, the average NBA player's salary during the 2002-03 season was \$4.54 million, and the average major league baseball player's salary at the start of the 2003 season was \$2.56 million. According to the Bureau of Labor Statistics, the mean annual wage in the US in 2002 was \$35,560.

[3] The Corporate Library の調査によれば、2002 年における S&P 500 の CEO に対する給料、ボーナス、株の配当とストックオプション等を含めた支払い金額の中央値は 365 万ドルであった。『Sports Illustrated』誌によれば、NBA のプレーヤーの 2002 年~2003 年のシーズンの給料の平均は 454 万ドル、そしてメジャーリーグ選手の給料の 2003 年シーズン初期の平均は 256 万ドルだ。一方、労働統計局によれば、米国の個人に対する平均

給与は 35,560 ドルである。

[4] In the early empire the price of an ordinary adult slave seems to have been about 2,000 sesterii (e.g. Horace, *Sat.* ii.7.43). A servant girl cost 600 (Martial vi.66), while Columella (iii.3.8) says that a skilled vine-dresser was worth 8,000. A doctor, P. Decimus Eros Merula, paid 50,000 sesterii for his freedom (Dessau, *Inscriptiones* 7812). Seneca (*Ep.* xxvii.7) reports that one Calvisius Sabinus paid 100,000 sesterii apiece for slaves learned in the Greek classics. Pliny (*Hist. Nat.* vii.39) says that the highest price paid for a slave up to his time was 700,000 sesterii, for the linguist (and presumably teacher) Daphnis, but that this had since been exceeded by actors buying their own freedom.

[4] 帝国の初期では、通常の大人の奴隷一人は 2,000 セステルティ程度であった(例：Horace, *Sat.* ii.7.43)。奴隷の少女の場合は 600 セステルティ (Martial vi.66) だったのに対し、コルメラ(iii.3.8)は、熟練したワイン製造の下働きなら 8,000 セステルティであったと述べている。医師 P. Decimus Eros Merula は自己の自由のために 5 万セステルティを支払った(Dessau, *Inscriptiones* 7812)。セネカ(*Ep.* xxvii.7)は、サビニのカルウィシウス家の一人はギリシャの古典に通じた奴隷一人に対し、10 万セステルティ支払ったと記録している。プリニー(*Hist. Nat.* vii.39)は、彼の時代に奴隷に支払われた最高の金額は 70 万セステルティで、それは言語学者(また、おそらく教師でもあった)Daphnis に対してであったが、またその後、役者が自分の自由のために支払う金額のほうが多くなったと述べている。

Classical Athens saw a similar variation in prices. An ordinary laborer was worth about 125 to 150 drachmae. Xenophon (*Mem.* ii.5) mentions prices ranging from 50 to 6,000 drachmae (for the manager of a silver mine).

古代アテネでも、同様な価格のばらつきが見られた。通常の労働者は 125 から 150 ドラクマであった。クセノポン(*Mem.* ii.5)は、価格は 50 から 6,000 ドラクマまで幅があったと述べている(最高額は銀山の管理者だった)。

For more on the economics of ancient slavery see:

古代の奴隷制度の経済に関しては、次を参照されたい。

Jones, A. H. M., "Slavery in the Ancient World," *Economic History Review*, 2:9 (1956), 185-199, reprinted in Finley, M. I. (ed.), *Slavery in Classical Antiquity*, Heffer, 1964.

[5] Eratosthenes (276—195 BC) used shadow lengths in different cities to estimate the Earth's circumference. He was off by only about 2%.

[5] エラトステネス(276～195 BC)は2つの都市における影の長さの違いから地球の半径を計算した。彼が出した数値の誤差はわずか2%ほどであった。

[6] No, and Windows, respectively.

[6] 答えはそれぞれ、否、および Windows。

[7] One of the biggest divergences between the Daddy Model and reality is the valuation of hard work. In the Daddy Model, hard work is in itself deserving. In reality, wealth is measured by what one delivers, not how much effort it costs. If I paint someone's house, the owner shouldn't pay me extra for doing it with a toothbrush.

[7] 父親モデルと現実とが最も異なるのは、頑張りの評価だ。父親モデルでは、頑張りはそれ自体が報酬の対象となる。現実では、富は結果で測られ、どれだけ頑張ったかは関係ない。もし私が誰かの家のペンキを塗るとして、歯ブラシを使って頑張っても余分にお金は貰えないだろう。

It will seem to someone still implicitly operating on the Daddy Model that it is unfair when someone works hard and doesn't get paid much. To help clarify the matter, get rid of everyone else and put our worker on a desert island, hunting and gathering fruit. If he's bad at it he'll work very hard and not end up with much food. Is this unfair? Who is being unfair to him?

父親モデルを無意識に信じてる人にとって、とても頑張った人があまり貰えないのは不公平に感じられるだろう。問題をより明確にするために、他の人間を全部取り除いて、この労働者が無人島にいて、狩りと果物の収集をやっているとしよう。彼がそういうことが苦手だとしたら、非常に頑張っても、大した食糧は手に入らないだろう。これは不公平だろうか。誰が彼に対して公平でない行いをしている？

[8] Part of the reason for the tenacity of the Daddy Model may be the dual meaning of "distribution." When economists talk about "distribution of income," they mean statistical distribution. But when you use the phrase frequently, you can't help associating it with the other sense of the word (as in e.g. "distribution of alms"), and thereby subconsciously seeing wealth as something that flows from some central tap. The word "regressive" as applied to tax rates has a similar effect, at least on me; how can anything *regressive* be good?

[8] 父親モデルがなかなか振り払えない理由のひとつは、分配(distribution)という言葉の意味の二重性かもしれない。経済学者が「収入の分散(distribution of income)」と言う場合、それは単に統計上の分散を指している。だが、この語句を頻繁に耳にしていると、それが単語のもうひとつの意味(「義援金の分配(distribution of alms)」のような)と結び付いてしまうんだ。そして、富が何か中心になる蛇口から流れてくるようなものと無意識に感じてしまう。私にとっては、税率における逆進性(regressive)という単語も似たような感覚になっている。後退(regressive)が良いことになるなんて。

[9] "From the beginning of the reign Thomas Lord Roos was an assiduous courtier of the young Henry VIII and was soon to reap the rewards. In 1525 he was made a Knight of the Garter and given the Earldom of Rutland. In the thirties his support of the breach with Rome, his zeal in crushing the Pilgrimage of Grace, and his readiness to vote the death-penalty in the succession of spectacular treason trials that punctuated Henry's erratic matrimonial progress made him an obvious candidate for grants of monastic property."

[9] 「若きヘンリー8世の統治の初めより、トーマス・ロード・ルースは勤勉な廷臣であり、ほどなく報償を得ることになる。1525年に彼はガーターのナイトとなり、ラトランドを伯爵領として与えられた。1530年代には、ローマとの戦いを支持したこと、恩寵の巡礼の乱の鎮圧への熱意、そしてヘンリー8世の度重なる離婚や再婚の際に継続して行われた、反逆者の審判において積極的に死刑票を投じたことなどから、彼は僧院の財産を譲渡される有力な候補となった。」

Stone, Lawrence, *Family and Fortune: Studies in Aristocratic Finance in the Sixteenth and Seventeenth Centuries*, Oxford University Press, 1973, p. 166.

[10] There is archaeological evidence for large settlements earlier, but it's hard to say what was happening in them.

[10] もっと初期に大きな開拓地があったことを示す考古学的な証拠はあるが、そこに何が起きたのかはよく分かっていない。

Hodges, Richard and David Whitehouse, *Mohammed, Charlemagne and the Origins of Europe*, Cornell University Press, 1983.

[11] William Cecil and his son Robert were each in turn the most powerful minister of the crown, and both used their position to amass fortunes among the largest of their times. Robert in particular took bribery to the point of treason. "As Secretary

of State and the leading advisor to King James on foreign policy, [he] was a special recipient of favour, being offered large bribes by the Dutch not to make peace with Spain, and large bribes by Spain to make peace." (Stone, *op. cit.*, p. 17.)

[11] ウィリアム・セシルとその息子ロバートは順に王の最有力の閣僚となり、その地位を使って彼らの時代に比類ない財産を蓄積するに至った。ロバートは特に、裏切りとも言える賄賂を受け取っていた。「主要閣僚であり、またジェームズ王の外交に関する首席アドバイザーとして、[彼]は特に贈賄の対象となった。オランダからは、スペインと講和しないために多額の賄賂を受け取り、一方でスペインからは講和するために多額の賄賂を受け取っていたのだ」(Stone, *op. cit.*, p. 17.)

[12] Though Balzac made a lot of money from writing, he was notoriously improvident and was troubled by debts all his life.

[12] バルザックは著作から多額の金を稼いでいたが、彼はおそろしく儉約心を欠いており、生涯借金に悩まされた。

[13] A Timex will gain or lose about .5 seconds per day. The most accurate mechanical watch, the Patek Philippe 10 Day Tourbillon, is rated at -1.5 to +2 seconds. Its retail price is about \$220,000.

[13] TIMEX は 1 日に 0.5 秒ほど進んだり遅れたりする。最も正確な機械式腕時計である、パテックフィリップの 10 Day Tourbillon は、-1.5 秒から+2 秒とされている。その小売価格は 22 万ドルだ。

[14] If asked to choose which was more expensive, a well-preserved 1989 Lincoln Town Car ten-passenger limousine (\$5,000) or a 2004 Mercedes S600 sedan (\$122,000), the average Edwardian might well guess wrong.

[14] 状態の良い 1989 年式のリンカーン 10 人乗りリムジン(5,000 ドル)と 2004 年式メルセデス S600 セダン(122,000 ドル)のどちらを選ぶか、とエドワード朝時代の普通の人に聞いたら、たぶん間違っただろうを選ぶだろう。

[15] To say anything meaningful about income trends, you have to talk about real income, or income as measured in what it can buy. But the usual way of calculating real income ignores much of the growth in wealth over time, because it depends on a consumer price index created by bolting end to end a series of numbers that are only locally accurate, and that don't include the prices of new inventions until they become so common that their prices stabilize.

[15] 収入の変化について何か意味のあることを言おうとするなら、本当の収入、つまりそ

れで何を買うかをもとにした収入で比較しなければならない。だが、実質収入を計算する普通の方法は、部分的にしか正確でない数値の端と端をくっつけたような小売価格指数をもとにしているため、時間による富の増加の大部分を無視しており、さらに新しい発明の価値については、それが広まって価格が安定してからでないと反映されない。

So while we might think it was very much better to live in a world with antibiotics or air travel or an electric power grid than without, real income statistics calculated in the usual way will prove to us that we are only slightly richer for having these things.

だから、私たちの感覚は、抗生剤や飛行機旅行や送電システムがある世界はそれがない世界よりもずっと良いとしているのに、普通の方法で計算された実質収入統計は、それらのものがあることでほんの少ししか裕福になっていないとするのだ。

Another approach would be to ask, if you were going back to the year x in a time machine, how much would you have to spend on trade goods to make your fortune? For example, if you were going back to 1970 it would certainly be less than \$500, because the processing power you can get for \$500 today would have been worth at least \$150 million in 1970. The function goes asymptotic fairly quickly, because for times over a hundred years or so you could get all you needed in present-day trash. In 1800 an empty plastic drink bottle with a screw top would have seemed a miracle of workmanship.

別のアプローチは、もし西暦 X 年にタイムマシンで戻れるとして、財産をつくるのにどれだけの価格の物を交換しなければならないかを考えることだ。例えば 1970 年に戻るなら、それは 500 ドル以下だろう。現在 500 ドルで買える計算機のパワーは、1970 年には 1 億 5000 万ドル出さなければ買えなかった。この関数は非常に速く限界値に漸近する。100 年を超えれば、現在ごみとされるものだけで必要なものは全て手に入れられるようになる。1800 年には、蓋付きの空のペットボトルは職人の奇跡のように思われていただろう。

[16] Some will say this amounts to the same thing, because the rich have better opportunities for education. That's a valid point. It is still possible, to a degree, to buy your kids' way into top colleges by sending them to private schools that in effect hack the college admissions process.

[16] 両者はどちらも同じ理由だという意見もあるだろう。裕福であれば、より良い教育を受けさせられるからだ。確かに、それはもっともな議論だ。ある程度までは、私立学校に子供をやることでトップクラスの大学への道を買うことはできる。これは実質的に、大学試験のプロセスをハックしていることになる。

According to a 2002 report by the National Center for Education Statistics, about 1.7% of American kids attend private, non-sectarian schools. At Princeton, 36% of the class of 2007 came from such schools. (Interestingly, the number at Harvard is significantly lower, about 28%.) Obviously this is a huge loophole. It does at least seem to be closing, not widening.

2002 年の米国教育統計センターの報告書によれば、米国の子供の 1.7%が宗教学校でない私立学校に通っている。プリンストン大学では、2007 年卒業のクラスのうち 36%がそういった学校の出身者である(面白いことに、ハーバードでは若干数値が低く、28%である)。明らかにこれは巨大な抜け穴である。だが、それは広がるのではなく縮まりつつあるようだ。

Perhaps the designers of admissions processes should take a lesson from the example of computer security, and instead of just assuming that their system can't be hacked, measure the degree to which it is.

大学入試のシステムの設計者は、コンピュータセキュリティの例から学ぶべきかもしれない。システムがハックされないと信じるのではなく、どのくらいハックされにくいかを常に評価するんだ。

Chapter 8 A PLAN FOR SPAM

第 8 章 スパムへの対策

August 2002

(This article describes the spam-filtering techniques used in the spamproof web-based mail reader we built to exercise [Arc](#). An improved algorithm is described in [Better Bayesian Filtering](#).)

(この記事は私たちが [Arc](#) を使って書いているスパム防止機能付き web ベースメールリーダーに使われているスパムフィルタリング技術を解説したものである。より改善されたアルゴリズムに関してはベイジアンフィルタの改善で述べている。)

I think it's possible to stop spam, and that content-based filters are the way to do it. The Achilles heel of the spammers is their message. They can circumvent any other barrier you set up. They have so far, at least. But they have to deliver their message, whatever it is. If we can write software that recognizes their messages, there is no way they can get around that.

スパムを止めることは可能だ。そして内容に基づいたフィルタリングがその方法だ。スパムを送る連中のアキレス腱は、彼らのメッセージだ。あなたがどんな障壁を築こうとも、彼らはそれをすり抜ける方法を見つける。少なくとも今まではそうだった。だが彼らは、何であれメッセージを届けなければならない。彼らのメッセージを認識できるソフトウェアを書くことができれば、彼らの道を絶つことができる。

To the recipient, spam is easily recognizable. If you hired someone to read your mail and discard the spam, they would have little trouble doing it. How much do we have to do, short of AI, to automate this process?

メールを受け取った人にとって、スパムを識別するのはたやすい。誰かを雇って自分宛に届くメールのスパムだけ捨ててくれるように頼んだとしても、きっとうまくやってくれるだろう。人工知能をまだ手にしていない現在、その手続きはどこまで自動化できるだろうか。

I think we will be able to solve the problem with fairly simple algorithms. In fact, I've found that you can filter present-day spam acceptably well using nothing

more than a Bayesian combination of the spam probabilities of individual words. Using a slightly tweaked (as described below) Bayesian filter, we now miss less than 5 per 1000 spams, with 0 false positives.

私は、この問題はかなり単純なアルゴリズムで解決できるんじゃないかと思っている。実際、個々の単語のスパム確率を単純にベイズの結合確率で計算するだけで、現在のスパムのかかなりの部分を取り除けることがわかった。これから述べるような、ちょっとした改良を加えたベイジアンフィルタでは、**1000** 通のスパム中で見逃すスパムが **5** 通以下で、しかも誤検出は **0** 通というところまで来ている。

The statistical approach is not usually the first one people try when they write spam filters. Most hackers' first instinct is to try to write software that recognizes individual properties of spam. You look at spams and you think, the gall of these guys to try sending me mail that begins "Dear Friend" or has a subject line that's all uppercase and ends in eight exclamation points. I can filter out that stuff with about one line of code.

スパムフィルタを書こうとする人はあまり統計的手法を最初に試したとらない。多くのハッカーは当初、直観的に、スパムの個々の特徴を認識するソフトウェアを書こうとする。スパムを眺めてこう思う。ふむ、この厚かましい連中は本文が"Dear Friend"で始まるか、全部大文字で最後に"!"が **8** つくらい続く表題を付けてメールを送ってきやがるな。一行コードを書けばそのくらいは取り除けるさ。

And so you do, and in the beginning it works. A few simple rules will take a big bite out of your incoming spam. Merely looking for the word "click" will catch 79.7% of the emails in my spam corpus, with only 1.2% false positives.

そしてやってみると、確かに最初のうちはうまく動く。ほんの **2~3** の簡単な規則でもって、スパムの大部分は除けるんだ。例えば"click"という単語を探すだけで、私のスパムコーパスの **79.7%**は捕まえられるし、誤検出は **1.2%**だ。

I spent about six months writing software that looked for individual spam features before I tried the statistical approach. What I found was that recognizing that last few percent of spams got very hard, and that as I made the filters stricter I got more false positives.

私は統計的手法を試す前に、スパムの個々の特徴を抽出するソフトウェアを書くのに **6** ヶ月ばかり費やした。そして分かったことは、最後の数%のスパムを捕まえるのはとても大変だということと、規則を厳しくすればするほど誤検出が増えてしまうということだった。

False positives are innocent emails that get mistakenly identified as spams. For most users, missing legitimate email is an order of magnitude worse than receiving spam, so a filter that yields false positives is like an acne cure that carries a risk of death to the patient.

誤検出とは、無害なメールなのにスパムと認識されてしまうメールのことだ。多くのユーザにとって、正しいメールを失うことはスパムを受け取るよりも何十倍もダメージが大きい。誤検出をしてしまうフィルタは、まるで死の危険のあるニキビ薬みたいなものだ。

The more spam a user gets, the less likely he'll be to notice one innocent mail sitting in his spam folder. And strangely enough, the better your spam filters get, the more dangerous false positives become, because when the filters are really good, users will be more likely to ignore everything they catch.

受け取るスパムが増えれば増えるほど、誤検出でスパムフォルダに行ってしまった無害なメールにユーザは気付きにくくなる。そして奇妙なことに、スパムフィルタの性能が良くなればなるほど、誤検出の危険は増していくのだ。というのは、フィルタが十分に良ければユーザはそれに信頼を置いて、それがスパムと判断したものは無視するようになるからだ。

I don't know why I avoided trying the statistical approach for so long. I think it was because I got addicted to trying to identify spam features myself, as if I were playing some kind of competitive game with the spammers. (Nonhackers don't often realize this, but most hackers are very competitive.) When I did try statistical analysis, I found immediately that it was much cleverer than I had been. It discovered, of course, that terms like "virtumundo" and "teens" were good indicators of spam. But it also discovered that "per" and "FL" and "ff0000" are good indicators of spam. In fact, "ff0000" (html for bright red) turns out to be as good an indicator of spam as any pornographic term.

なぜ私はなかなか統計的手法を試さなかったんだろう。たぶん、自分でスパムの特徴を見つけるゲームにはまっていたんだと思う。それはスパム業者との一種の競争だった(ハッカーでない人は知らないかもしれないが、たいていのハッカーは非常に競争好きだ)。統計的手法を試してみたらすぐに、私はそれが自分よりずっと賢いことに気付いた。統計的フィルタはもちろん"virtumundo"とか"teens"がスパムの良い指標だということを見つけたが、また"per"や"FL"や"ff0000"もそうだということも見つけたのだ。実際、"ff0000"(htmlメールで真っ赤の文字を表示するのに使われる)はポルノ用語と同じくらい、スパムの良い指標なのだ。

-- --

Here's a sketch of how I do statistical filtering. I start with one corpus of spam and one of nonspam mail. At the moment each one has about 4000 messages in it. I scan the entire text, including headers and embedded html and javascript, of each message in each corpus. I currently consider alphanumeric characters, dashes, apostrophes, and dollar signs to be part of tokens, and everything else to be a token separator. (There is probably room for improvement here.) I ignore tokens that are all digits, and I also ignore html comments, not even considering them as token separators.

では私が使っている統計的フィルタリングの概略を説明しよう。まず、スパムメールとスパムでないメールをそれぞれ集めたコーパスを作る。今のところどちらにも 4000 通くらいのメッセージがある。次に各集合中の全てのテキストを、ヘッダも本文中の `html` も `javascript` もいっしょくたにスキャンする。今のところは、アルファベット、数字、ダッシュ、アポストロフィ、そしてドルマークをトークンの構成要素とみなしている(おそらくここは改善の余地があるだろう)。全て数字のトークンと、`html` のコメントは無視する。後者はトークンの区切りとも見なさない。

I count the number of times each token (ignoring case, currently) occurs in each corpus. At this stage I end up with two large hash tables, one for each corpus, mapping tokens to number of occurrences.

次に各トークンがそれぞれの集合に現われた回数を数える。(今のところ大文字小文字は無視している)。ここまでで、トークンから出現回数を得ることのできる大きなハッシュテーブルが、両集合に対して得られたことになる。

Next I create a third hash table, this time mapping each token to the probability that an email containing it is a spam, which I calculate as follows [1]:

次に、各トークンから、そのトークンが含まれるメールがスパムである確率を得る第 3 のハッシュテーブルを次のようにして計算する。[1]:

```
(let ((g (* 2 (or (gethash word good) 0)))  
      (b (or (gethash word bad) 0)))  
(unless (< (+ g b) 5)  
  (max .01  
    (min .99 (float (/ (min 1 (/ b nbad))
```

$$\begin{aligned} & (+ (\min 1 (/ g \text{ ngood})) \\ & (\min 1 (/ b \text{ nbad}))))))))) \end{aligned}$$

where `word` is the token whose probability we're calculating, `good` and `bad` are the hash tables I created in the first step, and `ngood` and `nbad` are the number of nonspam and spam messages respectively.

ここで、`word` は確率を計算する単語、`good` と `bad` は第一ステップで作ったハッシュテーブル、そして `ngood` と `nbad` はそれぞれスパムでないメールとスパムメールの総数である。

I explained this as code to show a couple of important details. I want to bias the probabilities slightly to avoid false positives, and by trial and error I've found that a good way to do it is to double all the numbers in `good`. This helps to distinguish between words that occasionally do occur in legitimate email and words that almost never do. I only consider words that occur more than five times in total (actually, because of the doubling, occurring three times in nonspam mail would be enough). And then there is the question of what probability to assign to words that occur in one corpus but not the other. Again by trial and error I chose .01 and .99. There may be room for tuning here, but as the corpus grows such tuning will happen automatically anyway.

ここでコードを見せたのは、いくつかの重要な詳細を説明したかったからだ。まず、私は誤検出を避ける方向にバイアスをかけたかったので、試行錯誤してみた結果、`good` 中の単語の生起回数を 2 倍にするのがよいことが分かった。これは、正当なメール中にたまに出現し得る単語と、ほとんど全く出現しない単語とを区別するのに役立つ。また全体で 5 回以上出現していない単語は計算から外している(実際は正当なメール中の単語は倍にしているので、正当なメール中に 3 回単語が現われていれば考慮されることになる)。一方の集合にのみ現われる単語の確率はどうするかという問題もあるが、これも試行錯誤から 0.01 と 0.99 とした。ここには調整の余地があるかもしれないが、集合が大きくなればそのような調整は自動的に行われるだろう。

The especially observant will notice that while I consider each corpus to be a single long stream of text for purposes of counting occurrences, I use the number of emails in each, rather than their combined length, as the divisor in calculating spam probabilities. This adds another slight bias to protect against false positives. 注意深い人なら、私が単語の生起回数を数えるのには各集合をひと続きの長いテキストストリームと見做しているのに対し、分母のほうにはメールの長さの合計ではなくメールの数を使っていることに気付いただろう。これも誤検出を避ける方向へのバイアスとして働

く。

When new mail arrives, it is scanned into tokens, and the most interesting fifteen tokens, where interesting is measured by how far their spam probability is from a neutral .5, are used to calculate the probability that the mail is spam. If `probs` is a list of the fifteen individual probabilities, you calculate the [combined](#) probability thus:

新しいメールが届くと、それはトークンへと分解され、最も特徴的な 15 トークンが抽出され、それを基にメールがスパムである確率が計算される。ここで特徴的とは、その単語のスパム確率が 0.5 から遠く離れていることとする。`probs` を 15 のトークンそれぞれの確率とした場合、その結合確率は次のように計算される。

```
(let ((prod (apply #'* probs)))  
  (/ prod (+ prod (apply #'* (mapcar #'(lambda (x)  
                                         (- 1 x))  
                                         probs))))))
```

One question that arises in practice is what probability to assign to a word you've never seen, i.e. one that doesn't occur in the hash table of word probabilities. I've found, again by trial and error, that .4 is a good number to use. If you've never seen a word before, it is probably fairly innocent; spam words tend to be all too familiar.

もうひとつ、一度も見たことのない単語、すなわち単語の確率のハッシュテーブルに登録されていない単語に対していくら確率を割り当てるかという問題がある。これも試行錯誤から、0.4 が良いということが分かった。一度もその単語を見たことがなければ、その単語は無害なものであることが多い。スパムに出てくる単語はどれもどこかで見たようなものばかりだからだ。

There are examples of this algorithm being applied to actual emails in an appendix at the end.

付録にいくつかこのアルゴリズムを実際のメールに適用した例をあげておく。

I treat mail as spam if the algorithm above gives it a probability of more than .9 of being spam. But in practice it would not matter much where I put this threshold, because few probabilities end up in the middle of the range.

上記のアルゴリズムが 0.9 以上の確率を返した場合、私はそのメールをスパムと判断する。

実際にはこの閾値はあまり問題ではない。確率が中くらいになるメールはほとんどないからだ。

One great advantage of the statistical approach is that you don't have to read so many spams. Over the past six months, I've read literally thousands of spams, and it is really kind of demoralizing. Norbert Wiener said if you compete with slaves you become a slave, and there is something similarly degrading about competing with spammers. To recognize individual spam features you have to try to get into the mind of the spammer, and frankly I want to spend as little time inside the minds of spammers as possible.

統計的手法の良いところのひとつは、たくさんのスパムに目を通さなくてもよいことだ。過去 6 ヶ月にわたって私は文字通り何千というスパムを読んできたが、げんなりするような体験だった。ノーバート・ウィーナーは奴隷と競争するものは奴隷になると言ったが、スパム業者と競争するのもそれに似た点がある。スパムの特徴を捉えようとするには、スパム業者の思考の中に入っていこうとする必要があるが、正直に言って私はできる限りスパム業者の思考からは離れていきたいのだ。

But the real advantage of the Bayesian approach, of course, is that you know what you're measuring. Feature-recognizing filters like SpamAssassin assign a spam "score" to email. The Bayesian approach assigns an actual probability. The problem with a "score" is that no one knows what it means. The user doesn't know what it means, but worse still, neither does the developer of the filter. How many *points* should an email get for having the word "sex" in it? A probability can of course be mistaken, but there is little ambiguity about what it means, or how evidence should be combined to calculate it. Based on my corpus, "sex" indicates a .97 probability of the containing email being a spam, whereas "sexy" indicates .99 probability. And Bayes' Rule, equally unambiguous, says that an email containing both words would, in the (unlikely) absence of any other evidence, have a 99.97% chance of being a spam.

だがベイズ確率を使った方法の本当の利点は、自分が何を計測しているかがはっきりしていることだ。SpamAssassin のような特徴認識型のフィルタは各メールに「スパムスコア」を与えようとする。ベイズ確率を使った方法は、実際の確率を与える。「スコア」の問題点は、誰もそれが本当は何を意味しているのか分からない点だ。ユーザには当然分からない

し、もっと悪いことに、開発者にもわからない。例えばメールに"sex"という語が入っていたとして、それに何ポイント与えるべきだ?確率ももちろん間違った使われ方をすることはあるが、少なくともそれが何を意味しているかに関して曖昧さはほとんどなく、それぞれの証拠を結合する計算法も確立している。私のメールアーカイブにおいては、"sex"は、それを含んでいるメールがスパムである確率が 0.97 であることを示しているし、"sexy"なら 0.99 になる。そしてベイズの規則を用いれば、他の証拠がない場合(普通はそんなことはないが)、両方の単語を含んでいるメールがスパムである確率は 99.97%であることが確かに言えるわけだ。

Because it is measuring probabilities, the Bayesian approach considers all the evidence in the email, both good and bad. Words that occur disproportionately *rarely* in spam (like "though" or "tonight" or "apparently") contribute as much to decreasing the probability as bad words like "unsubscribe" and "opt-in" do to increasing it. So an otherwise innocent email that happens to include the word "sex" is not going to get tagged as spam.

確率を計測するために、ベイズ確率を使った方法ではスパムである証拠とスパムでない証拠を両方とも考慮しなければならない。"unsubscribe"や"opt-in"といった単語がスパムである確率を押し上げるのと同じくらい、スパム中にほとんど現われない、"though"、"tonight"、"apparently"といった単語はその確率を押し下げる。したがって、正当なメールがたまたま"sex"という語を含んでいたとしてもそれがスパムと認識されることはない。

Ideally, of course, the probabilities should be calculated individually for each user. I get a lot of email containing the word "Lisp", and (so far) no spam that does. So a word like that is effectively a kind of password for sending mail to me. In my earlier spam-filtering software, the user could set up a list of such words and mail containing them would automatically get past the filters. On my list I put words like "Lisp" and also my zipcode, so that (otherwise rather spammy-sounding) receipts from online orders would get through. I thought I was being very clever, but I found that the Bayesian filter did the same thing for me, and moreover discovered of a lot of words I hadn't thought of.

理想的には、この確率はユーザごとに計算しておくべきだ。私は"Lisp"という単語が含まれたメールをたくさん受け取るが、(少なくとも現在までは)"Lisp"を含むスパムを受け取ったことはない。こういった単語は、私にメールを送るための一種のパスワードにもなるわけだ。私が書いた初期のスパムフィルタリングソフトでは、ユーザがそういった単語を登録できるようになっていて、それらの単語を含んでいるメールはフィルタを通らずに届くようにしていた。私自身のリストには"Lisp"と私の郵便番号を登録しておき、オンラインで注

文した品物のレシート(それ自体はスパムによく似ている)が私の元に届くようにしていた。その時は素晴らしいアイデアだと思っていた。でもベイジアンフィルタは同じことをやってのけ、さらに私自身が気付きもしなかった多くの他の単語も見つけ出してくれたのだ。

When I said at the start that our filters let through less than 5 spams per 1000 with 0 false positives, I'm talking about filtering my mail based on a corpus of my mail. But these numbers are not misleading, because that is the approach I'm advocating: filter each user's mail based on the spam and nonspam mail he receives. Essentially, each user should have two delete buttons, ordinary delete and delete-as-spam. Anything deleted as spam goes into the spam corpus, and everything else goes into the nonspam corpus.

この記事の最初で、私たちのフィルタは 1000 通中スパムを通すのは 5 つ以下で、しかも誤検出は 0 だと書いた。これは、私のメールの集合を用いて私のメールを処理した場合の話だ。だがそれは的外れな数字ではない。そういう使い方こそを私は強く推奨しているんだ。それぞれのユーザが、自分が受け取ったスパムとスパム以外のメールの集合を基にしてフィルタリングを行うんだ。基本的なアイデアは、ユーザメールに削除ボタンを 2 種類設けるというものだ。通常の削除とスパムとして削除だ。スパムとして削除されたものはスパム集合に追加され、それ以外のメールは正当な集合に追加される。

You could start users with a seed filter, but ultimately each user should have his own per-word probabilities based on the actual mail he receives. This (a) makes the filters more effective, (b) lets each user decide their own precise definition of spam, and (c) perhaps best of all makes it hard for spammers to tune mails to get through the filters. If a lot of the brain of the filter is in the individual databases, then merely tuning spams to get through the seed filters won't guarantee anything about how well they'll get through individual users' varying and much more trained filters.

最初に初期値となるフィルタを与えておいたとしても、いずれ各ユーザは自分が受け取ったメールを元にした単語ごとのスパム確率のテーブルを持つことになる。これによって(a)フィルタがよりうまく働くようになり(b)各ユーザが自分がスパムと判断する基準を正確に持てるようになり、(c)何よりも、スパム業者にとってフィルタを回避するメールを作成することがずっと困難になる。フィルタの判断基準の多くが個人ごとのデータベースに依存するとしたら、スパムを初期フィルタの設定を回避するようにしただけのスパムは、ユーザ毎に学習されたフィルタを通ることを保証できない。

Content-based spam filtering is often combined with a whitelist, a list of senders

whose mail can be accepted with no filtering. One easy way to build such a whitelist is to keep a list of every address the user has ever sent mail to. If a mail reader has a delete-as-spam button then you could also add the from address of every email the user has deleted as ordinary trash.

内容を基にしたスパムフィルタリングはしばしばホワイトリスト、つまりフィルタが無条件で回避される差出人のリストと組み合わせて使われることが多い。そのようなホワイトリストを作成する簡単な方法のひとつは、ユーザが今まで送ったことのあるメールアドレスを保存しておくことだ。もしメールリーダーが「スパムとして削除」ボタンを持っていれば、さらにユーザがスパムでないごみとして捨てたメールの差出人アドレスも使うことができる。

I'm an advocate of whitelists, but more as a way to save computation than as a way to improve filtering. I used to think that whitelists would make filtering easier, because you'd only have to filter email from people you'd never heard from, and someone sending you mail for the first time is constrained by convention in what they can say to you. Someone you already know might send you an email talking about sex, but someone sending you mail for the first time would not be likely to. The problem is, people can have more than one email address, so a new from-address doesn't guarantee that the sender is writing to you for the first time. It is not unusual for an old friend (especially if he is a hacker) to suddenly send you an email with a new from-address, so you can't risk false positives by filtering mail from unknown addresses especially stringently.

私はホワイトリストの使用も推奨しているが、それはフィルタの性能を良くするためというより、計算負荷を減らすためだ。以前は、ホワイトリストを使うことでフィルタリングが簡単になると思っていた。今まで一度もメールをやりとりしたことの無いアドレスからのメールだけをフィルタにかければよいし、初めてのメールのやりとりの内容には決まりきった点が多々あるからだ。あなたが既に知っている人はもしかするとセックスに関する話題をメールしてくるかもしれないが、初めてあなたにメールを送ろうという人がセックスについて語ることはあまりありそうにない。しかしこの仮定には問題があった。人はよく複数のメールアドレスを持つので、差出人アドレスを初めて見るというだけではあなたに初めてメールを送る人かどうかの判断はできないのだ。古くからの友人が(特に彼がハッカーである場合)突然新しいアドレスからメールをよこすのは珍しいことではない。したがって知らないアドレスからというだけでスパムと誤検出してしまうようなことはぜひとも避けたい。

In a sense, though, my filters do themselves embody a kind of whitelist (and

blacklist) because they are based on entire messages, including the headers. So to that extent they "know" the email addresses of trusted senders and even the routes by which mail gets from them to me. And they know the same about spam, including the server names, mailer versions, and protocols.

もちろん、ある意味では私のフィルタは自分で一種のホワイトリストを(そしてブラックリストも)実現していると言えるだろう。ヘッダを含めた全てのメッセージを対象にしているからだ。私のフィルタは信頼できる差出人のアドレスや、その配送経路さえも「知っている」。同じく、スパムを送るサーバ名、メーラーのバージョン、プロトコルさえも。

If I thought that I could keep up current rates of spam filtering, I would consider this problem solved. But it doesn't mean much to be able to filter out most present-day spam, because spam evolves. Indeed, most [antispam techniques](#) so far have been like pesticides that do nothing more than create a new, resistant strain of bugs.

もし私が現在のスパムフィルタリング率を将来にわたって保持できるとしたなら、この問題は解決したと言えるだろう。しかし、現時点でのスパムを排除できたというだけではそうは言えない。スパムは進化するからだ。実際、多くのアンチスパム技術は、より抵抗力の強い虫を作り出すだけの殺虫剤としてしか働いてこなかった。

I'm more hopeful about Bayesian filters, because they evolve with the spam. So as spammers start using "c0ck" instead of "cock" to evade simple-minded spam filters based on individual words, Bayesian filters automatically notice. Indeed, "c0ck" is far more damning evidence than "cock", and Bayesian filters know precisely how much more.

私はベイジアンフィルタについてはずっと大きな希望を持っている。ベイジアンフィルタはそれ自身がスパムと共に進化するからだ。例えばスパム業者が単純なスパムフィルタを回避するために"cock"の代わりに"c0ck"という語を使い始めたとしたら、ベイジアンフィルタは自動的にそれに気付くだろう。実際、"c0ck"は"cock"よりもずっと明確なスパムの証拠であり、それがどのくらいより明確かということさえもベイジアンフィルタは知っている。

Still, anyone who proposes a plan for spam filtering has to be able to answer the question: if the spammers knew exactly what you were doing, how well could they

get past you? For example, I think that if checksum-based spam filtering becomes a serious obstacle, the spammers will just switch to mad-lib techniques for generating message bodies.

それでも、スパムフィルタリング手法を提案する人は次の質問に答えられなければならない。もしスパム業者があなたのアルゴリズムを正確に知っていたとして、やつらはどれだけそれをすり抜けられるだろうか。例えばチェックサムを使ったスパムフィルタが大きな障害になったとすれば、スパム業者は **mad-lib** を使ってメッセージを生成するようになるだろう。

To beat Bayesian filters, it would not be enough for spammers to make their emails unique or to stop using individual naughty words. They'd have to make their mails indistinguishable from your ordinary mail. And this I think would severely constrain them. Spam is mostly sales pitches, so unless your regular mail is all sales pitches, spams will inevitably have a different character. And the spammers would also, of course, have to change (and keep changing) their whole infrastructure, because otherwise the headers would look as bad to the Bayesian filters as ever, no matter what they did to the message body. I don't know enough about the infrastructure that spammers use to know how hard it would be to make the headers look innocent, but my guess is that it would be even harder than making the message look innocent.

ベイジアンフィルタを破るには、スパム業者はメールを特殊なものにしたり怪しい単語を使うのを止めるだけでは不十分だ。普通のメールと区別がつかないようにスパムを作らなくてはならなくなる。そして、これは彼らの活動を厳しく制限するだろうと私は思う。スパムのほとんどは物売り込む文句だから、あなたが通常のメールとして物売り込む文句を常に受け取っているのでもない限り、どうしても正当なメールとは異なる特徴を持ってしまう。さらにスパム業者たちはシステムを変え続けなければならない。メッセージ本文をいくら工夫しても、ベイジアンフィルタにヘッダを検出されたらおしまいだ。私はスパム業者がどんなシステムを使っているか十分に知らないで、ヘッダを無難なものに細工するのがどのくらい彼らにとって難しいかは分からないのだが、おそらくメッセージ本文を無害なものに装うよりも困難なのではないか。

Assuming they could solve the problem of the headers, the spam of the future will probably look something like this:

スパム業者たちがヘッダの問題を回避できたとすると、おそらく将来のスパムはこんな感じになってしまうはずだ。

Hey there. Thought you should check out the following:

面白いサイトを見つけたよ。見てみて。

<http://www.27meg.com/foo>

because that is about as much sales pitch as content-based filtering will leave the spammer room to make. (Indeed, it will be hard even to get this past filters, because if everything else in the email is neutral, the spam probability will hinge on the url, and it will take some effort to make that look neutral.)

売り込み文句が内容を基にしたフィルタリングをすり抜ける余地はこのくらいしか残されていない。(実際はこれでもフィルタを通り抜けるのは難しいかもしれない。というのは、他の全ての要素が中立ならば URL のスパム確率が効いてくるはずで、したがって URL も中立であるかのように装わねばならない。)

Spammers range from businesses running so-called opt-in lists who don't even try to conceal their identities, to guys who hijack mail servers to send out spams promoting porn sites. If we use filtering to whittle their options down to mails like the one above, that should pretty much put the spammers on the "legitimate" end of the spectrum out of business; they feel obliged by various state laws to include boilerplate about why their spam is not spam, and how to cancel your "subscription," and that kind of text is easy to recognize.

スパム業者も、自分の身元を隠そうともしない、いわゆるオプトインリストを使っているビジネスから、サーバを乗っ取ってポルノサイトの宣伝を送りつけるのまで、さまざまだ。もしフィルタリングが上のようなスパムしか通さないようになったとすると、おそらく「合法的」領域のスパム業者はやってゆけなくなるだろう。彼らはさまざまな州法によって彼らのメールがなぜスパムではないか、そしてどうやれば「購読」を停止できるか、等を書き連ねなければならず、それらの文章は容易に認識できるからだ。

(I used to think it was naive to believe that stricter laws would decrease spam. Now I think that while stricter laws may not decrease the amount of spam that spammers *send*, they can certainly help filters to decrease the amount of spam that recipients actually see.)

(私は以前はより厳しい法律がスパムを減らすだろうと無邪気に信じていた。今では、厳しい法律ではスパム業者が送り出すスパムの量は減らせないだろうと思っている。が、法律のおかげでフィルタは有効に働き、受け取り人が目にするスパムの量は確かに減るだろう。)

All along the spectrum, if you restrict the sales pitches spammers can make, you

will inevitably tend to put them out of business. That word *business* is an important one to remember. The spammers are businessmen. They send spam because it works. It works because although the response rate is abominably low (at best 15 per million, vs 3000 per million for a catalog mailing), the cost, to them, is practically nothing. The cost is enormous for the recipients, about 5 man-weeks for each million recipients who spend a second to delete the spam, but the spammer doesn't have to pay that.

どんな種類のスパム業者であれ、もし私たちが彼らの売り文句をこのように制限できるならば、いずれ彼らはビジネスをやめざるを得ないだろう。ビジネスという言葉に注目してほしい。スパム業者はビジネスマンなんだ。彼らがスパムを送り出すのは、それが有効だからだ。スパムへの反応は信じられないほど少ないが(カタログ送付の反応が 100 万通あたり 3000 通であるのに対し、スパムは良くて 100 万通あたり 15 通というところ)、スパムを送るコストはほとんどかからない。受け取り人には、100 万人がスパムを削除するのに 1 秒かけたとして総計 5 人週ものコストを強いているが、スパム業者がそれを払っているわけではない。

Sending spam does cost the spammer something, though. [2] So the lower we can get the response rate-- whether by filtering, or by using filters to force spammers to dilute their pitches-- the fewer businesses will find it worth their while to send spam.

それでもスパム業者にはいくばくかのコストがかかる。[2] だから、フィルタリングによって、あるいはフィルタリング回避のために効果的な売り文句が使えなくなるかして、反応率が低くなれば、ビジネス的なうまみは減っていくわけだ。

The reason the spammers use the kinds of [sales pitches](#) that they do is to increase response rates. This is possibly even more disgusting than getting inside the mind of a spammer, but let's take a quick look inside the mind of someone who *responds* to a spam. This person is either astonishingly credulous or deeply in denial about their sexual interests. In either case, repulsive or idiotic as the spam seems to us, it is exciting to them. The spammers wouldn't say these things if they didn't sound exciting. And "thought you should check out the following" is just not going to have nearly the pull with the spam recipient as the kinds of things that spammers say now. Result: if it can't contain exciting sales pitches, spam becomes less effective as a marketing vehicle, and fewer businesses want to use it.

スパム業者が売り文句を使うのは、それが返信率を上げるからだ。スパム業者の思考に入

っていくよりもさらに気色悪いかもしれないが、スパムに返信してしまう人間の思考をちょっと覗いてみよう。おそらくその人物は、信じられないほどだまされやすいか、性的な関心を極端に抑圧しているのであろう。どちらにせよ、私たちが見て嫌悪感を感じたり間抜けに見えたりするスパムが、彼らにとっては興奮するものなのだろう。そうでなければスパム業者がそういう言葉を使うわけがないからだ。すると、「このリンク先を見てみて」式の文句は、現在のスパムが使っている売り文句ほどには、スパムに返信してしまう人間の心を惹きつけそうにない。結果として、売り文句を含めることができなければ、スパムはマーケティング方法としては効率の悪いものとなり、それを使おうとするビジネスも減っていくだろう。

That is the big win in the end. I started writing spam filtering software because I didn't want have to look at the stuff anymore. But if we get good enough at filtering out spam, it will stop working, and the spammers will actually stop sending it.

最後にそうなれば、私たちの大きな勝利だ。私は、自分がスパムをもう見たくないからスパムフィルタを書き始めた。しかしスパムフィルタの技術が十分に進歩すれば、スパムの効果はなくなり、スパム業者たちもそれを送るのをやめるだろう。

— — —

Of all the approaches to fighting spam, from software to laws, I believe Bayesian filtering will be the single most effective. But I also think that the more different kinds of antispam efforts we undertake, the better, because any measure that constrains spammers will tend to make filtering easier. And even within the world of content-based filtering, I think it will be a good thing if there are many different kinds of software being used simultaneously. The more different filters there are, the harder it will be for spammers to tune spams to get through them.

ソフトウェアから法律まで、スパムと戦う全てのアプローチのうち、ベイジアンフィルタリングは最も効果的なものであると私は思っている。しかし私たちがさまざまな種類の反スパム活動をすればするほど、状況は良くなるだろう。それらはスパム業者に制約を課し、結果としてフィルタリングを容易にするからだ。そして内容に基づくフィルタリングの中だけでも、さまざまな種類のソフトウェアが同時に使われていくのが好ましい。フィルタの種類が多ければ多いほど、そのどれをも通過するスパムを作り出すのは困難になるからだ。

Appendix: Examples of Filtering

付録：フィルタリングの例

[Here](#) is an example of a spam that arrived while I was writing this article. The fifteen most interesting words in this spam are:

これはこの記事を書いている最中に私の元に届いたスパムの例だ。このスパムの最も特徴的な 15 単語は次の通りだ。

qvp0045
indira
mx-05
intimail
\$7500
freeyankeedom
cdo
bluefoxmedia
jpg
unsecured
platinum
3d0
qves
7c5
7c266675

The words are a mix of stuff from the headers and from the message body, which is typical of spam. Also typical of spam is that every one of these words has a spam probability, in my database, of .99. In fact there are more than fifteen words with probabilities of .99, and these are just the first fifteen seen.

これらの単語はヘッダからのものとメッセージ本文からのものが混ざっているが、スパムにはよくあることだ。また、これもスパムによくあることだが、上の全ての単語は私のデータベースではスパム確率 0.99 を与えられている。実はこのメッセージには確率 0.99 の単語は 15 個以上含まれていて、上のものはただ最初の 15 個だったというだけなのだ。

Unfortunately that makes this email a boring example of the use of Bayes' Rule. To

see an interesting variety of probabilities we have to look at [this](#) actually quite atypical spam.

残念ながらこのメールはベイズの規則の使用例としてはおもしろいものではない。確率がうまい具合に散っているものとしては、これも典型的なスパムだがこういう例がある。

The fifteen most interesting words in this spam, with their probabilities, are:

このスパムの特徴的な 15 単語とそのスパム確率は次の通りだ。

madam	0.99
promotion	0.99
republic	0.99
shortest	0.047225013
mandatory	0.047225013
standardization	0.07347802
sorry	0.08221981
supported	0.09019077
people's	0.09019077
enter	0.9075001
quality	0.8921298
organization	0.12454646
investment	0.8568143
very	0.14758544
valuable	0.82347786

This time the evidence is a mix of good and bad. A word like "shortest" is almost as much evidence for innocence as a word like "madam" or "promotion" is for guilt. But still the case for guilt is stronger. If you combine these numbers according to Bayes' Rule, the resulting probability is .9027.

今回は良い証拠と悪い証拠が混ざっている。"madam"や"promotion"が有罪の有力な証拠であるのと同じくらい、"shortest"といった単語は無罪の有力な証拠だ。しかしこの場合でもまだ有罪の方が強力である。これらの数字をベイズの規則で組み合わせれば、結果の確率は 0.9027 となる。

"Madam" is obviously from spams beginning "Dear Sir or Madam." They're not very common, but the word "madam" *never* occurs in my legitimate email, and it's all about the ratio.

"Madam"はあきらかに、"Dear Sir or Madam"で始まっているスパムから来たものだ。この文句自体はそれほど多く見られるものではないが、"madam"は私の正当なメール中には1回も出て来ない。そのためこれだけの確率となった。

"Republic" scores high because it often shows up in Nigerian scam emails, and also occurs once or twice in spams referring to Korea and South Africa. You might say that it's an accident that it thus helps identify this spam. But I've found when examining spam probabilities that there are a lot of these accidents, and they have an uncanny tendency to push things in the right direction rather than the wrong one. In this case, it is not entirely a coincidence that the word "Republic" occurs in Nigerian scam emails and this spam. There is a whole class of dubious business propositions involving less developed countries, and these in turn are more likely to have names that specify explicitly (because they aren't) that they are republics.[\[3\]](#)

"Republic"が高いスコアを得ているのは、それがしばしばナイジェリア絡みの詐欺のメールに現われるからだ。また韓国や南アフリカのものも1~2通あった。それだけでこの単語がスパムと判定されるのに使われるのは偶然だと思うかもしれない。だが、スパム確率をよく調べてみると、このような偶然は非常に多くあり、それらが不思議と性能を悪いほうよりは良いほうへと押し上げているのだ。この場合、"Republic"がナイジェリア絡みの詐欺のスパムとこのスパムとの両方に現われたのは必ずしも偶然とは言えない。発展途上国を舞台にした怪しいビジネスの案内はスパムの1ジャンルを作っていて、それらの国名には"Republic(共和国)"が明示的に含まれていることが多いからだ(何故なら、実際には共和制ではないからだ)。[\[3\]](#)

On the other hand, "enter" is a genuine miss. It occurs mostly in unsubscribe instructions, but here is used in a completely innocent way. Fortunately the statistical approach is fairly robust, and can tolerate quite a lot of misses before the results start to be thrown off.

一方、"enter"は本当のミスだ。確かにこの単語は購読停止の案内に現われることが多いが、このメールの場合は全く罪のない用法で使われている。幸いなことに統計的なアプローチは頑健で、かなり多くの誤りがあっても結果がおかしくなることはない。

For comparison, [here](#) is an example of that rare bird, a spam that gets through the filters. Why? Because by sheer chance it happens to be loaded with words that occur in my actual email:

比較のために、フィルタを通り抜けた非常に稀なスパムとしてこれを紹介しておこう。な

ぜか?このスパムは本当にたまたま、私の正当なメールに含まれる単語を大量に含んでいたのだ。

perl	0.01
python	0.01
tcl	0.01
scripting	0.01
morris	0.01
graham	0.01491078
guarantee	0.9762507
cgi	0.9734398
paul	0.027040077
quite	0.030676773
pop	30.042199217
various	0.06080265
prices	0.9359873
managed	0.06451222
difficult	0.071706355

There are a couple pieces of good news here. First, this mail probably wouldn't get through the filters of someone who didn't happen to specialize in programming languages and have a good friend called Morris. For the average user, all the top five words here would be neutral and would not contribute to the spam probability.

ここにも 2 つばかり、良いニュースはある。まず、このメールはたぶん、プログラミング言語を専門にしており、かつ **Morris** という友人がいる人でない限り、フィルタを通ることはないだろう。平均的なユーザにとってはこの最初の 5 単語は中立の語であり、スパム確率には貢献しないだろう。

Second, I think filtering based on word pairs (see below) might well catch this one: "cost effective", "setup fee", "money back" -- pretty incriminating stuff. And of course if they continued to spam me (or a network I was part of), "Hostex" itself would be recognized as a spam term.

第 2 に、単語のペアによるフィルタリングを使えば(後述)、このメールも捕まえられるかもしれない。"cost effective"、"setup fee"、"money back"——いかにもスパムの臭いがする。そして彼らが私に(あるいは私が参加しているネットワークに)スパムを送り続けていれば、"Hostex"という語自体がスパム語として認識されるようになるだろう。

Finally, [here](#) is an innocent email. Its fifteen most interesting words are as follows:

最後に、正当なメールの例をあげておこう。このメールの特徴的な 15 語は次の通りだ。

continuation	0.01
describe	0.01
continuations	0.01
example	0.033600237
programming	0.05214485
i'm	0.055427782
examples	0.07972858
color	0.9189189
localhost	0.09883721
hi	0.116539136
california	0.84421706
Same	0.15981844
Spot	0.1654587
us-ascii	0.16804294
what	0.19212411

Most of the words here indicate the mail is an innocent one. There are two bad smelling words, "color" (spammers love colored fonts) and "California" (which occurs in testimonials and also in menus in forms), but they are not enough to outweigh obviously innocent words like "continuation" and "example".

ほとんどの単語は無害なものだ。2 つばかり、臭い単語がある。"color"(スパム業者は色付き文字が大好きだ)と"california"(保証書とフォームのメニューによく現われる)だ。しかしそれらも、明らかに無害の単語である"continuation"や"example"を負かすには至らない。

It's interesting that "describe" rates as so thoroughly innocent. It hasn't occurred in a single one of my 4000 spams. The data turns out to be full of such surprises. One of the things you learn when you analyze spam texts is how narrow a subset of the language spammers operate in. It's that fact, together with the equally characteristic vocabulary of any individual user's mail, that makes Bayesian filtering a good bet.

"describe"がほぼ無害になっているのは興味深い。私の 4000 通のスパムの中の 1 通たり

とも、この単語を含んではいなかった。このような驚くデータがたくさんある。スパムテキストを分析していてわかることのひとつは、スパム業者の使う単語がどれだけ狭いかということだ。だからこそ、個人のメールに現われる同様に特徴的な単語と組み合わせることにより、ベイジアンフィルタリングが効果的な方法となり得ているのだ。

Appendix: More Ideas

付録：他のアイディア

One idea that I haven't tried yet is to filter based on word pairs, or even triples, rather than individual words. This should yield a much sharper estimate of the probability. For example, in my current database, the word "offers" has a probability of .96. If you based the probabilities on word pairs, you'd end up with "special offers" and "valuable offers" having probabilities of .99 and, say, "approach offers" (as in "this approach offers") having a probability of .1 or less. 私がまた試していないアイディアのひとつは、個々の単語だけではなく単語のペア、あるいは 3 単語の組み合わせでフィルタリングを行うことだ。そうすればさらに鋭い確率の評価ができるだろう。例えば、私の現在のデータベースでは、"offers"は 0.96 の確率が付いている。単語のペアを使えば、おそらく"special offers"や"valuable offers"は 0.99 となり、一方で"approach offers"("this approach offers"と言った句に出てくる)はおそらく 0.1 かそれ以下の確率となるだろう。

The reason I haven't done this is that filtering based on individual words already works so well. But it does mean that there is room to tighten the filters if spam gets harder to detect. (Curiously, a filter based on word pairs would be in effect a Markov-chaining text generator running in reverse.)

まだ私がこれを試していないのは、個々の単語だけで十分に良い結果を得ているからだ。しかしスパムの検出がより難しくなってきた時、フィルタの性能をさらに上げる余地があるということだ。(ちなみに、単語のペアに基づいたフィルタは、実質的にマルコフ連鎖のテキストジェネレータを逆に走らせているのと同じだ。)

Specific spam features (e.g. not seeing the recipient's address in the to: field) do of course have value in recognizing spam. They can be considered in this algorithm by treating them as virtual words. I'll probably do this in future versions, at least for a handful of the most egregious spam indicators. Feature-recognizing spam filters are right in many details; what they lack is an overall discipline for combining evidence.

スパムの個々の特徴(例えば受け取り人のアドレスが **To:** フィールドにない)もちろんスパムを認識するのに役立つ。そのような特徴は、このアルゴリズム中で仮想単語として実装できるだろう。将来のバージョンでは、最もはなはだしいスパムの特徴のいくつかを組み込んでみるつもりだ。特徴抽出型のスパムフィルタは各論では正しい。それに欠けているのは、得られた証拠を組み合わせるための原理だ。

Recognizing nonspam features may be more important than recognizing spam features. False positives are such a worry that they demand extraordinary measures. I will probably in future versions add a second level of testing designed specifically to avoid false positives. If a mail triggers this second level of filters it will be accepted even if its spam probability is above the threshold.

スパムでない特徴を見つけるのは、スパムである特徴を見つけるのよりも重要だ。誤検出は重大であり、特に慎重にならなければならない。私は将来のバージョンに、誤検出を避けるために特別にデザインした第 2 段階のテストを組み込むつもりだ。メールが閾値以上のスパム確率を持っていても、それが第 2 段階のフィルタにひっかかったら、それはスパムではないと認識される。

I don't expect this second level of filtering to be Bayesian. It will inevitably be not only ad hoc, but based on guesses, because the number of false positives will not tend to be large enough to notice patterns. (It is just as well, anyway, if a backup system doesn't rely on the same technology as the primary system.)

第 2 段階のフィルタリングにはおそらくベイズ確率は使わないだろう。たぶんそれはアドホックなだけでなく、推測によるルールにならざるを得ないだろう。誤検出はパターンを見付けられるようになるほど多くはないからだ。(いずれにせよ、バックアップシステムには主システムとは異なる技術を用いておくのは良いことだろう。)

Another thing I may try in the future is to focus extra attention on specific parts of the email. For example, about 95% of current spam includes the url of a site they want you to visit. (The remaining 5% want you to call a phone number, reply by email or to a US mail address, or in a few cases to buy a certain stock.) The url is in such cases practically enough by itself to determine whether the email is spam. もうひとつ私が試したいと思っているのは、メールの特定の部分に注目することだ。例えば、現在のスパムのうち **95%** はあなたに訪れて欲しい **URL** を載せている(残りの **5%** は電話番号か、メールでの返信か、郵送の住所か、またまれに特定の株を買う、なんてのもある)。その場合、その **URL** を認識できればそのメールがスパムであると決定するのに十分だ。

Domain names differ from the rest of the text in a (non-German) email in that they often consist of several words stuck together. Though computationally expensive in the general case, it might be worth trying to decompose them. If a filter has never seen the token "xxxporn" before it will have an individual spam probability of .4, whereas "xxx" and "porn" individually have probabilities (in my corpus) of .9889 and .99 respectively, and a combined probability of .9998.

ドメイン名は、(ドイツ語以外の)メールの他のテキストと違った特徴を持っている。いくつかの単語がくっついていることが多いという特徴だ。一般的に計算するには計算量が多くなってしまうだろうが、それらを個々の単語に分解してみるのは試してみる価値はあるかもしれない。フィルタが"xxxporn"という単語を一度も見たことがなかったとするとそれに確率 0.4 をつけてしまうだろうが、"xxx"と"porn"はそれぞれ(私のデータでは)0.9899 と 0.99 の確率を持っており、その組み合わせ確率は 0.9998 だ。

I expect decomposing domain names to become more important as spammers are gradually forced to stop using incriminating words in the text of their messages. (A url with an ip address is of course an extremely incriminating sign, except in the mail of a few sysadmins.)

スパム業者がメッセージ中に怪しい単語を避けるようになるにつれ、ドメイン名を分解するのは重要になってゆくと私は思っている。(IP アドレスの URL は、システム管理者宛のメールでなければ、真っ黒だと言えるだろう。)

It might be a good idea to have a cooperatively maintained list of urls promoted by spammers. We'd need a trust metric of the type studied by Raph Levien to prevent malicious or incompetent submissions, but if we had such a thing it would provide a boost to any filtering software. It would also be a convenient basis for boycotts. スпам業者が宣伝する URL のリストを協力して作成してゆくのは良い考えかもしれない。ただし、悪意ある登録や不正確な登録を防ぐための、Raph Levien が研究したような信頼の指標が必要だろう。だが、もしそういうものが得られれば、フィルタリングソフトの有効性は一気に高まる。それはまたボイコットをするためにも便利だろう。

Another way to test dubious urls would be to send out a crawler to look at the site before the user looked at the email mentioning it. You could use a Bayesian filter to rate the site just as you would an email, and whatever was found on the site could be included in calculating the probability of the email being a spam. A url that led to a redirect would of course be especially suspicious.

怪しい URL を調べるもうひとつの方法は、ユーザがその URL を含むメールを見る前にロボ

ットに覗きに行かせるというものだ。そしてメールに適用するのと同じようにサイトのスコアをベイジアンフィルタを使って計算し、それをメールのスパム確率の計算に含ませることができるだろう。リダイレクトされる URL などは特に怪しい。

One cooperative project that I think really would be a good idea would be to accumulate a giant corpus of spam. A large, clean corpus is the key to making Bayesian filtering work well. Bayesian filters could actually use the corpus as input. But such a corpus would be useful for other kinds of filters too, because it could be used to test them.

協力してやるプロジェクトとして非常に良いと私が考えるのは、巨大なスパムコーパスを蓄積してゆくことだ。ベイジアンフィルタがうまく動く鍵は、大きく綺麗なコーパスにある。そういうコーパスはベイジアンフィルタの入力としても使えるだけでなく、その他のフィルタにもテスト用として利用できる。

Creating such a corpus poses some technical problems. We'd need trust metrics to prevent malicious or incompetent submissions, of course. We'd also need ways of erasing personal information (not just to-addresses and ccs, but also e.g. the arguments to unsubscribe urls, which often encode the to-address) from mails in the corpus. If anyone wants to take on this project, it would be a good thing for the world.

そのようなコーパスを作るにはいくつか技術的な問題がある。悪意ある登録や不正確な登録を防ぐための信頼の指標はもちろん必要だ。また、個人情報を消去する方法も要る(ただ **To:** と **Cc:** を消すだけでなく、例えば登録削除要請の URL の引数にもしばしば **to:** アドレスがエンコードされて含まれている)。誰かがこのプロジェクトを始めたら、それは世界にとって良いものとなるだろう。

Appendix: Defining Spam

付録：スパムの定義

I think there is a rough consensus on what spam is, but it would be useful to have an explicit definition. We'll need to do this if we want to establish a central corpus of spam, or even to compare spam filtering rates meaningfully.

スパムとは何であるかについてだいたい合意はあると思うが、明確な定義を与えておくことも役に立つだろう。集約されたスパムのコーパスを作ったり、あるいはスパムフィルタのフィルタ率の意味ある比較をするためには明確な定義が必要になる。

To start with, spam is not unsolicited commercial email. If someone in my neighborhood heard that I was looking for an old Raleigh three-speed in good condition, and sent me an email offering to sell me one, I'd be delighted, and yet this email would be both commercial and unsolicited. The defining feature of spam (in fact, its *raison d'etre*) is not that it is unsolicited, but that it is automated.

まず、スパムとは、頼まないのに送ってくる商用のメールではない、というところから始めよう。例えば私が古いレイリーの 3 段変速自転車の状態のいいやつを欲しがっているのを私の近所の人が耳にはさんで、手持ちのを売ってもいいよというメールを送ってきたとしたら、このメールは商用でありかつ頼んでいないわけだが、それでも私は喜ぶだろう。スパムを定義づけている特徴は、頼んでいないのに送ってくるのではなく、自動的に送ってくることなのだ(そしてそれがスパムの存在意義でもある)。

It is merely incidental, too, that spam is usually commercial. If someone started sending mass email to support some political cause, for example, it would be just as much spam as email promoting a porn site.

また、スパムに商用の内容が多いのも、たまたまそうなかだけだと言える。例えば誰かが何かの政治運動に関するメールを大量に送り始めたとしたら、それはポルノサイトの宣伝と同じくらいスパムである。

I propose we define spam as **unsolicited automated email**. This definition thus includes some email that many legal definitions of spam don't. Legal definitions of spam, influenced presumably by lobbyists, tend to exclude mail sent by companies that have an "existing relationship" with the recipient. But buying something from a company, for example, does not imply that you have solicited ongoing email from them. If I order something from an online store, and they then send me a stream of spam, it's still spam.

私はスパムの定義として、頼んでいないのに自動的に送ってくるメール、を提案する。この定義は、法的にはスパムに当たらないメールも含んでいる。おそらくロビイストに影響を受けたせいで、法的なスパムの定義は受け取り人と既に関係がある会社から送られてくるメールはスパムに含まれないものとすることが多い。しかし、例えば何かをその会社から買うだけではそれ以降ずっとメールを受け取ることを頼んだことにはならない。もし私がオンラインで何かを注文して、相手の会社がその後継続してスパムを送り付けてきたら、それはやっぱりスパムなのだ。

Companies sending spam often give you a way to "unsubscribe," or ask you to go to their site and change your "account preferences" if you want to stop getting

spam. This is not enough to stop the mail from being spam. Not opting out is not the same as opting in. Unless the recipient explicitly checked a clearly labelled box (whose default was no) asking to receive the email, then it is spam.

スパムを送ってくる会社はしばしば購読停止を行う方法を提供したり、スパムを止めたければ会社のサイトにいてあなたのアカウントのオプションを変更するように書いてくる。だがこれだけではそのメールがスパムでないとするには足りない。オプトアウトでないということはオプトインであるということと同等にはならない。受け取り人が、メールを受け取る意志があると明示されたチェックボックス(デフォルトで **off** になっているもの)をクリックしたというのでなければ、それはスパムなのだ。

In some business relationships, you do implicitly solicit certain kinds of mail. When you order online, I think you implicitly solicit a receipt, and notification when the order ships. I don't mind when Verisign sends me mail warning that a domain name is about to expire (at least, if they are the actual registrar for it). But when Verisign sends me email offering a FREE Guide to Building My E-Commerce Web Site, that's spam.

ビジネスの関係によっては、暗黙のうちに何らかのメールを送ってくれることを頼んでいる場合がある。例えばオンラインで注文をすれば、レシートや発送のお知らせを送ってくることを暗黙のうちに依頼したと考えてよいだろう。私は、ベリサインが私のドメインネームがもうすぐ期限切れになるというメールを送ってきたって気にしない(少なくとも彼らが本当のレジストラならね)。だがベリサインが e コマースウェブサイト構築の無料ガイドの提供のお知らせをメールしてきたら、それはスパムだ。

Notes

注釈

[1] The examples in this article are translated into Common Lisp for, believe it or not, greater accessibility. The application described here is one that we wrote in order to test a new Lisp dialect called [Arc](#) that is not yet released.

[1] この記事の例は、(信じてくれないかもしれないが)より多くの人にわかりやすいように Common Lisp に書き直したものだ。ここで述べたアプリケーションは新しい Lisp の方言 Arc をテストするために書いたものだが、Arc 自体はまだリリースされていない。

[2] Currently the lowest rate seems to be about \$200 to send a million spams. That's very cheap, 1/50th of a cent per spam. But filtering out 95% of spam, for example, would increase the spammers' cost to reach a given audience by a factor

of 20. Few can have margins big enough to absorb that.

[2] 現在のところ、最も安いところで 100 万通あたり 200 ドルというのが相場のようなのだ。これは大変に安い。スパム 1 通あたり 50 分の 1 セントだ。だが、95%のスパムがフィルタされたとすると、ターゲットへとメールを届かせるためのスパム業者のコストは 20 倍に跳ね上がることになる。それだけ大きなマージンを確保しているスパム業者はあまりいないだろう。

[3] As a rule of thumb, the more qualifiers there are before the name of a country, the more corrupt the rulers. A country called The Socialist People's Democratic Republic of X is probably the last place in the world you'd want to live.

[3] おおざっぱな指針として、国の名前に付く修飾語が多ければ多いほど、その統治者は腐っているとと言えるだろう。何とか社会主義人民民主共和国というのはたぶん一番住みたくない国だ。

Thanks to Sarah Harlin for reading drafts of this; Daniel Giffin (who is also writing the production Arc interpreter) for several good ideas about filtering and for creating our mail infrastructure; Robert Morris, Trevor Blackwell and Erann Gat for many discussions about spam; Raph Levien for advice about trust metrics; and Chip Coldwell and Sam Steingold for advice about statistics.

Chapter 9 TASTE FOR MAKERS

第9章 ものづくりのセンス

February 2002



"...Copernicus' aesthetic objections to [equants] provided one essential motive for his rejection of the Ptolemaic system...."

- Thomas Kuhn, *The Copernican Revolution*

「[エクアントへの]美学的な反発が、コペルニクスに天動説を拒否させたひとつの重要な動機であった…」

- トーマス・クーン、『コペルニクス革命』

"All of us had been trained by Kelly Johnson and believed fanatically in his insistence that an airplane that looked beautiful would fly the same way."

- Ben Rich, *Skunk Works*

「我々は皆、ケリー・ジョンソンに教えを受けており、外見の美しい航空機は美しく飛ぶはずだという彼の主張を熱狂的に信じていた」

- ベン・R・リッチ、『スカンクワークス』

"Beauty is the first test: there is no permanent place in this world for ugly mathematics."

- G. H. Hardy, *A Mathematician's Apology*

「美が最初のテストだ。この世には醜い数学の生き残る余地はない」

- G・H・ハーディ、『一数学者の弁明』

I was talking recently to a friend who teaches at MIT. His field is hot now and every year he is inundated by applications from would-be graduate students. "A lot of them seem smart," he said. "What I can't tell is whether they have any kind of taste."

最近、MIT で教えている友人と話をした。彼の研究分野は今とても人気があり、毎年大学院への応募書類が殺到している。「みんな賢そうだよ」と彼は言った。「ただ、センスがあるかどうかを見分けるのは難しい」

Taste. You don't hear that word much now. And yet we still need the underlying concept, whatever we call it. What my friend meant was that he wanted students who were not just good technicians, but who could use their technical knowledge to design beautiful things.

センス。あるいは審美眼。最近この言葉をあまり聞かなくなった。だがこの言葉の指す概念の重要性が失われたわけではない。私の友人が言いたかったのは、技術者として優秀なだけでなく、その技術的な知識を使って美しいものをデザインできる学生が欲しいということだ。

Mathematicians call good work "beautiful," and so, either now or in the past, have scientists, engineers, musicians, architects, designers, writers, and painters. Is it just a coincidence that they used the same word, or is there some overlap in what they meant? If there is an overlap, can we use one field's discoveries about beauty to help us in another?

数学者は良い仕事を「美しい」と形容する。科学者、技術者、音楽家、建築家、デザイナー、作家、画家、そういった人々も、過去や現在に、「美しい」という形容詞を使ってきた。皆が同じ単語を使うのは単なる偶然なんだろうか。それともその意味するものの間に何かの共通項があるのだろうか。もし共通項があるとすれば、ひとつの分野での美に関する知見を他の分野にも使えないだろうか。

For those of us who design things, these are not just theoretical questions. If there is such a thing as beauty, we need to be able to recognize it. We need good taste to make good things. Instead of treating beauty as an airy abstraction, to be either blathered about or avoided depending on how one feels about airy

abstractions, let's try considering it as a practical question: *how do you make good stuff?*

ものをデザインする私たちのような立場にとって、これは単なる理論的な問題じゃない。美というものが存在するなら、私たちはそれをきっちり認識できなくちゃならない。良いものを作るには、良いセンス、美を見分ける力が必要なんだ。美を実体のない抽象概念として扱って、それについてたわごとを並べたり逆に避けて通ったりするのではなく、現実的な問題として考えてみようじゃないか。どうやったら良いものが作れる？

If you mention taste nowadays, a lot of people will tell you that "taste is subjective." They believe this because it really feels that way to them. When they like something, they have no idea why. It could be because it's beautiful, or because their mother had one, or because they saw a movie star with one in a magazine, or because they know it's expensive. Their thoughts are a tangle of unexamined impulses.

現代では、センスについて語ろうとすると、多くの人がこう言うだろう。「センスなんて主観的なものさ」と。実際彼らにとってはそう感じられるから、そう信じているんだ。何かを好ましいと思っても、なぜそうなのか彼らには分からない。それが美しいからなのかもしれないし、母親も同じ物をもっていたからなのかもしれない。雑誌で映画スターと一緒に写っていたのを見たからかもしれないし、あるいは単に高価なものだと知っているからかもしれない。頭の中では、いろいろな思い付きがよく考えられないまま絡み合っているんだ。

Most of us are encouraged, as children, to leave this tangle unexamined. If you make fun of your little brother for coloring people green in his coloring book, your mother is likely to tell you something like "you like to do it your way and he likes to do it his way."

私たちの多くは子供の頃に、この絡み合いを考えないままに教えられるように教えられてきた。弟が塗り絵で人を緑に塗っているのを馬鹿にしていると、母親はきっと「あなたにはあなたの好み、弟には弟の好みがあるのよ」とか何とか言うだろう。

Your mother at this point is not trying to teach you important truths about aesthetics. She's trying to get the two of you to stop bickering.

ここで母親は美学についての重要な真実を教えようとしているわけではない。単にあなたと弟が喧嘩するのを止めたいだけだ。

Like many of the half-truths adults tell us, this one contradicts other things they tell us. After dinning into you that taste is merely a matter of personal preference, they take you to the museum and tell you that you should pay attention because Leonardo is a great artist.

大人はこんなふうに、半分だけ正しいことをよく言う。でもそういうものはたいてい、大人が言う他のことと矛盾している。センスは単なる個人の好みだと子供のあなたにくどくど教え込んだ後で、大人はあなたを美術館に連れてゆき、レオナルド・ダ・ヴィンチは偉大な芸術家だからしっかり見ておきなさいなんて言うんだ。

What goes through the kid's head at this point? What does he think "great artist" means? After having been told for years that everyone just likes to do things their own way, he is unlikely to head straight for the conclusion that a great artist is someone whose work is *better* than the others'. A far more likely theory, in his Ptolemaic model of the universe, is that a great artist is something that's good for you, like broccoli, because someone said so in a book.

このとき、子供の頭にはどんな考えが浮かぶだろう。「偉大な芸術家」ってどんな意味だと思うだろう。人には人それぞれのやり方があるって何百回も言われた後では、偉大な芸術家は他の人より良い仕事をする人だ、という結論にはなかなかたどり着けない。どっちかという、子供の天動説的な世界の中では、「偉大な芸術家」っていうのはきっと自分にとって良いものなんだって説に落ち着くだろう。ちょうど、本にそう書いてあったからブロッコリーが身体にいいんだって考えるみたいに。

Saying that taste is just personal preference is a good way to prevent disputes. The trouble is, it's not true. You feel this when you start to design things.

センスは個人の好みだと言うのは論争を避けるには良い方法だ。問題は、それが真実でないということだ。ものをデザインし始めるとそれを感じることになる。

Whatever job people do, they naturally want to do better. Football players like to win games. CEOs like to increase earnings. It's a matter of pride, and a real pleasure, to get better at your job. But if your job is to design things, and there is no such thing as beauty, then there is *no way to get better at your job*. If taste is just personal preference, then everyone's is already perfect: you like whatever you like, and that's it.

どんな仕事をしていても、それをもっとうまくやりたいと思うようになるのは自然なことだ。フットボールの選手は試合に勝ちたいし、CEO は利益を増やしたい。仕事をより良くやることはプライドの問題でもあるし、本物の喜びでもある。でも、ものをデザインするのが仕事で、そして客観的な美というものが存在しないとしたら、仕事をもっとうまくやるということは原理的に不可能じゃないか。センスが単なる個人の好みなら誰もが既に完璧だ。みんな自分の好きなものを好きになり、それでオしまい。

As in any job, as you continue to design things, you'll get better at it. Your tastes will change. And, like anyone who gets better at their job, you'll know you're getting better. If so, your old tastes were not merely different, but worse. Poof goes the axiom that taste can't be wrong.

他のどんな仕事と同じように、ものをデザインする仕事を続けていけばだんだんうまくできるようになってくる。センスが変わってくるのだ。そして仕事の上達する人は誰も、自分が上達してくるのがわかる。だとすれば、あなたの古いセンスは単に違うというだけでなく、今より悪かったのだ。センスに正しいも間違いもないなんて公理を信じるのは腰抜けだけだ。

Relativism is fashionable at the moment, and that may hamper you from thinking about taste, even as yours grows. But if you come out of the closet and admit, at least to yourself, that there is such a thing as good and bad design, then you can start to study good design in detail. How has your taste changed? When you made mistakes, what caused you to make them? What have other people learned about design?

現代では相対主義が流行してて、自分のセンスが育っていたとしても、それについて考えるのは難しいかもしれない。でも眼を開けて、良いデザインと悪いデザインというものがあるのだと少なくとも自分の中で認めなければ、良いデザインについて詳しく学び始めることはできない。自分のセンスはどう変わった?何が原因で間違いを犯した?他の人はデザインについて何を学んだんだろう?

Once you start to examine the question, it's surprising how much different fields' ideas of beauty have in common. The same principles of good design crop up again and again.

この問題を考え始めると、さまざまな分野における美の概念の共通項の多さに驚くだろう。良いデザインに関して同じ原則が何度も何度も出てくるのだ。

Good design is simple. You hear this from math to painting. In math it means that a shorter proof tends to be a better one. Where axioms are concerned, especially, less is more. It means much the same thing in programming. For architects and designers it means that beauty should depend on a few carefully chosen structural elements rather than a profusion of superficial ornament. (Ornament is not in itself bad, only when it's camouflage on insipid form.) Similarly, in painting, a still life of a few carefully observed and solidly modelled objects will tend to be more interesting than a stretch of flashy but mindlessly repetitive painting of, say, a lace collar. In writing it means: say what you mean and say it briefly.

良いデザインは単純である。数学から絵画に至るまでこの文句を聞いたことがあるだろう。数学では、これは短い証明ほど良いものである傾向があるということだ。特に公理に関しては少ない方がよい。プログラミングにおいても同じことが言える。建築家とデザイナーにとっては、この原則は、美とは表面的な装飾で溢れさせることではなく、厳選された少数の要素から組み立てられるものだという事だ(装飾それ自体は悪いものではないが、凡庸さを誤魔化すものであってはならない)。絵画においても同様に、少数の素材をよく観察してしっかり型取った静物画のほうが、例えばレースの襟なんかを無意味に繰り返し描いたものよりは面白いものになるだろう。文章においてはこういうことだ。言いたいことを簡潔に言え。

It seems strange to have to emphasize simplicity. You'd think simple would be the default. Ornate is more work. But something seems to come over people when they try to be creative. Beginning writers adopt a pompous tone that doesn't sound anything like the way they speak. Designers trying to be artistic resort to swooshes and curlicues. Painters discover that they're expressionists. It's all evasion. Underneath the long words or the "expressive" brush strokes, there is not much going on, and that's frightening.

単純さをわざわざ強調するのを変に思うかもしれない。何もしないのが単純で、装飾を付けるのはもっと大変な仕事じゃないかと。だが、人々が創造的であろうとする時、何か不思議な力がやってくるんだ。駆け出しの小説家は普段話している言葉とかけ離れた大げさな言い回しを使いたがる。デザイナーは芸術的であろうとして、画家は表現主義者になる。これらはみな言い逃れだ。長ったらしい単語や「表現主義的な」絵のタッチの下には、実は大した中身がない。それが怖いんだ。

When you're forced to be simple, you're forced to face the real problem. When you

can't deliver ornament, you have to deliver substance.

単純でなければならぬと強制されれば、本当の問題と向き合わなければならぬ。装飾を見せられないなら、実体を見せなくちゃならない。

Good design is timeless. In math, every proof is timeless unless it contains a mistake. So what does Hardy mean when he says there is no permanent place for ugly mathematics? He means the same thing Kelly Johnson did: if something is ugly, it can't be the best solution. There must be a better one, and eventually someone will discover it.

良いデザインは永遠である。数学では間違いがない限り全ての証明は永遠だ。じゃあハーディが醜い数学は生き残らないと言った意味はなんだろう。それはケリー・ジョンソンが言ったのと同じことだ。醜いものは最良の解にはなり得ない。きっとより良い解があるはずで、いずれ誰かがそれを発見する。

Aiming at timelessness is a way to make yourself find the best answer: if you can imagine someone surpassing you, you should do it yourself. Some of the greatest masters did this so well that they left little room for those who came after. Every engraver since Durer has had to live in his shadow.

色褪せない解を目標にするのは最良の解を見つける良い方法だ。誰かが自分よりうまい方法を見つけるだろうと思っているくらいなら、その方法を自分で見つけなくちゃ。最も偉大な人々はこの方法を突き詰めたおかげで、後から来る人が改善する余地がほとんどないものを作り上げた。デュラー以降の版画家は彼の影の中で生きることを強いられたのだ。

Aiming at timelessness is also a way to evade the grip of fashion. Fashions almost by definition change with time, so if you can make something that will still look good far into the future, then its appeal must derive more from merit and less from fashion.

色褪せない解を目標にするのは、流行にはまるのを避ける良い方法でもある。その定義からして流行は時間とともに変化する。ずっと未来にも良く見えるものを作れたとすれば、それはそのものの真価が受け入れられたわけで、流行に乗ったからではない。

Strangely enough, if you want to make something that will appeal to future generations, one way to do it is to try to appeal to past generations. It's hard to guess what the future will be like, but we can be sure it will be like the past in

caring nothing for present fashions. So if you can make something that appeals to people today and would also have appealed to people in 1500, there is a good chance it will appeal to people in 2500.

ちょっとひねったアイディアだが、もし未来の世代にも気に入られるものを作りたいなら、過去の世代にも気に入られるものを作ろうとするのもひとつの方法だ。未来がどうなるかを予測するのは難しいが、現在の流行に左右されないという意味では過去も同じようなものだ。だから現在の人々に好まれて、しかも 1500 年の人々にも好まれたであろうものが作れば、それが 2500 年の人々にも好まれるであろう可能性は高い。

Good design solves the right problem. The typical stove has four burners arranged in a square, and a dial to control each. How do you arrange the dials? The simplest answer is to put them in a row. But this is a simple answer to the wrong question. The dials are for humans to use, and if you put them in a row, the unlucky human will have to stop and think each time about which dial matches which burner. Better to arrange the dials in a square like the burners.

良いデザインは正しい問題を解決する。よくある台所のレンジは 4 つのバーナーが四角に並んでいて、それぞれを調節するつまみが付いている。このつまみをどうやって配置したらいいだろう。一番単純な答えはそれを横一列に並べることだ。だがそれは、間違っただけに対する単純な答えなのだ。つまみは人が使うものだ。つまみが横一列に並んでいたら、不幸なユーザは毎回どのつまみがどのバーナーに対応しているかを考えなければならないだろう。バーナーと同じようにつまみを四角に配置したほうがよい。

A lot of bad design is industrious, but misguided. In the mid twentieth century there was a vogue for setting text in sans-serif fonts. These fonts *are* closer to the pure, underlying letterforms. But in text that's not the problem you're trying to solve. For legibility it's more important that letters be easy to tell apart. It may look Victorian, but a Times Roman lowercase g is easy to tell from a lowercase y. 悪いデザインの多くは、念入りであるが勘違いした仕事の結果だ。20 世紀中盤に、文章をサンセリフで組版するのが流行ったことがあった。確かに装飾なしのフォントの方がもとの文字の形に近い。しかし文章において、それは解くべき問題ではないのだ。読みやすさのためには文字同士が区別しやすいことの方が重要だ。ビクトリア王朝風に見えたとしても、Times Roman 書体の小文字の g は小文字の y と容易に区別がつく。

Problems can be improved as well as solutions. In software, an intractable

problem can usually be replaced by an equivalent one that's easy to solve. Physics progressed faster as the problem became predicting observable behavior, instead of reconciling it with scripture.

解と同じように、問題も改善してゆける。ソフトウェアでは、追跡不可能な問題を、等値でしかも解くのが容易な問題へと置き換えることができる場合が多い。物理学は、観察した事象を経典と調和させることが課題だった時代より、事象を予測することが課題になってからずっと速く進歩するようになった。

Good design is suggestive. Jane Austen's novels contain almost no description; instead of telling you how everything looks, she tells her story so well that you envision the scene for yourself. Likewise, a painting that suggests is usually more engaging than one that tells. Everyone makes up their own story about the Mona Lisa.

良いデザインは想像力を喚起する。ジェーン・オースティンの小説には叙述がほとんどない。いろんなものがどんなふうに見えるかを説明する代わりに、彼女は物語を実にうまく語るのだ、あなたは情景を目に浮かべることができるのだ。同様に、いろいろ想像できる絵画の方が、全部描いてあるものよりも見ていて飽きない。誰もがモナ・リザに関して自分だけの物語を作り上げる。

In architecture and design, this principle means that a building or object should let you use it how you want: a good building, for example, will serve as a backdrop for whatever life people want to lead in it, instead of making them live as if they were executing a program written by the architect.

この原則は建築とデザインにおいては、建物や品物は人々が使いたいように使えるべきであるということになる。例えば良い建物は、そこに住む人々が自分の生きたいように生活を送る背景となるべきであり、建築家の書いたプログラムを実行しているように生きるためであってはならない。

In software, it means you should give users a few basic elements that they can combine as they wish, like Lego. In math it means a proof that becomes the basis for a lot of new work is preferable to a proof that was difficult, but doesn't lead to future discoveries; in the sciences generally, citation is considered a rough indicator of merit.

ソフトウェアにおいては、ユーザが自分の思う通りにまるでレゴのように組み合わせるこ

とができる、少数の基本要素を提供すべきだ。数学においては、多くの新しい発見の基礎となるような証明の方が、難しいけれど新しい発見へとつながらない証明より望ましい。一般に科学では引用が真価の目安となる。

Good design is often slightly funny. This one may not always be true. But Durer's [engravings](#) and Saarinen's [womb chair](#) and the [Pantheon](#) and the original [Porsche 911](#) all seem to me slightly funny. Godel's incompleteness theorem seems like a practical joke.

良いデザインはしばしばちょっと滑稽だ。これはいつでも正しいというわけではないかもしれない。でも、デュラーの版画やサーリネンのウーム・チェア、そしてパンテオンや初代のポルシェ 911 は、私にとってはちょっとばかり滑稽に見える。ゲーデルの不完全性定理ときたらまるで手の込んだ悪ふざけみたいだ。

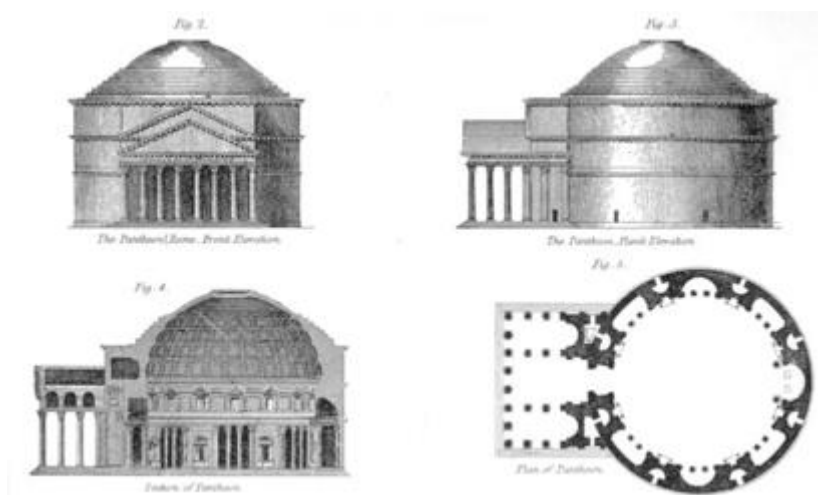


PILATE WASHING HIS HANDS

Albrecht Durer, engraving, 11.6 x 7.4 cm (yes, cm), 1512. [Click for a larger version.](#)



WOMB CHAIR
Eero Saarinen, 1947.



THE PANTHEON
From 1897 *Encyclopaedia Britannica*.



1974 PORSCHE 911 MODELS

I think it's because humor is related to strength. To have a sense of humor is to be strong: to keep one's sense of humor is to shrug off misfortunes, and to lose one's sense of humor is to be wounded by them. And so the mark-- or at least the prerogative-- of strength is not to take oneself too seriously. The confident will often, like swallows, seem to be making fun of the whole process slightly, as Hitchcock does in his films or Bruegel in his paintings-- or Shakespeare, for that matter.

私思うに、これはユーモアが力と関連しているからだ。ユーモアのセンスがあることは、力強くあることである。ユーモアのセンスを持ち続けければ不幸な出来事も軽く笑い飛ばせるが、ユーモアを失えばそういう出来事に傷ついてしまう。深刻に考えすぎないことは力強い人の特権だ。自信を持つ人は困難な過程でさえちょっとばかり茶化すようなところがある。ヒッチコックが彼の映画でやったように。あるいはブリューゲルが彼の絵画でやったように。この点に関してはシェークスピアも同様だ。

Good design may not have to be funny, but it's hard to imagine something that could be called humorless also being good design.

良いデザインが常に面白おかしくなければならぬというわけじゃない。でも、良いデザインでかつユーモアを感じないものを想像するのはとても難しい。

Good design is hard. If you look at the people who've done great work, one thing they all seem to have in common is that they worked very hard. If you're not

working hard, you're probably wasting your time.

良いデザインをするのは難しい。素晴らしい仕事をした人々を見ると、彼らに共通するのは並外れた努力であることが分かる。努力しないで美しいものを作ろうなんて、それは時間の無駄というものだ。

Hard problems call for great efforts. In math, difficult proofs require ingenious solutions, and those tend to be interesting. Ditto in engineering.

困難な問題は多大な努力を要求する。数学では難しい証明には技巧的な解法が必要で、それはまた興味深いものにもなる。工学でも同様だ。

When you have to climb a mountain you toss everything unnecessary out of your pack. And so an architect who has to build on a difficult site, or a small budget, will find that he is forced to produce an elegant design. Fashions and flourishes get knocked aside by the difficult business of solving the problem at all.

山を登る時は、不必要なものは荷物に入れないだろう。同じように、建築家は難しい建築物や限られた予算を前にすれば、エレガントなデザインを強いられることを悟る。流行や華やかな装飾は、そもそも問題を解くという困難な仕事の中に入って来る余地がないのだ。

Not every kind of hard is good. There is good pain and bad pain. You want the kind of pain you get from going running, not the kind you get from stepping on a nail. A difficult problem could be good for a designer, but a fickle client or unreliable materials would not be.

どんな頑張りでもよいというわけではない。良い苦しみと、悪い苦しみがある。走っている時に感じるような苦しみなら結構だが、釘を踏み抜いた時のようなひどい苦しみは御免だ。困難な問題はデザイナーにとって良いものだが、気まぐれな顧客や信頼できない素材はそうではない。

In art, the highest place has traditionally been given to paintings of people. There is something to this tradition, and not just because pictures of faces get to press buttons in our brains that other pictures don't. We are so good at looking at faces that we force anyone who draws them to work hard to satisfy us. If you draw a tree and you change the angle of a branch five degrees, no one will know. When you change the angle of someone's eye five degrees, people notice.

美術では、伝統的に最も価値を認められてきたのが人物画であった。これには理由がある。他の画が押さない、私たちの脳内のボタンを押すとかいうことだけではない。人間が人の顔を見分ける能力があまりに優れているため、人の顔の画を描く者は観る者を満足させる

ためには非常に努力をしなければならないのだ。木を描いて、枝の角度を 5 度ばかり変えたとしても、誰も気が付くまい。だが、誰かの目を 5 度傾けて描いたら、みんな気付くだろう。

When Bauhaus designers adopted Sullivan's "form follows function," what they meant was, form *should* follow function. And if function is hard enough, form is forced to follow it, because there is no effort to spare for error. Wild animals are beautiful because they have hard lives.

バウハウスのデザイナーがサリヴァンの「形態は機能に従う」という方針を採用した時、彼らが意図したのは「形態は機能に従うはずだ」ということだった。そして機能が難しいものであるなら、形態もそれに従わざるを得ない。誤りを犯す余地はないからだ。野性の動物が美しいのは、彼らが困難な生を生きているからだ。

Good design looks easy. Like great athletes, great designers make it look easy. Mostly this is an illusion. The easy, conversational tone of good writing comes only on the eighth rewrite.

良いデザインは簡単に見える。偉大な陸上選手のように、偉大なデザイナーはデザインをいとも簡単に見せる。ほとんどの場合、それは幻想だ。読みやすい会話調の良文を書くには、書き直しが 8 回は必要だ。

In science and engineering, some of the greatest discoveries seem so simple that you say to yourself, I could have thought of that. The discoverer is entitled to reply, why didn't you?

科学や工学では、いくつかの偉大な発見はあまりにも単純に見えて、「これなら自分でも思い付いたよ」と言いたくなることがある。発見者はこう言うだろう。「じゃあ君が思い付けばよかったじゃないか」

Some Leonardo heads are just a few lines. You look at them and you think, all you have to do is get eight or ten lines in the right place and you've made this beautiful portrait. Well, yes, but you have to get them in *exactly* the right place. The slightest error will make the whole thing collapse.

レオナルド・ダ・ヴィンチの肖像画のいくつかはほんの数本の線にすぎない。それを見て人はこう考えるかもしれない。美しい肖像画を描くには、正しい位置に 8 本か 10 本ばかりの線を置けばよいだけだと。確かにその通り。だが、それらの線は完璧に正しい位置に置

かれなければ。ほんのわずかの誤りが全てを駄目にしてしまう。

Line drawings are in fact the most difficult visual medium, because they demand near perfection. In math terms, they are a closed-form solution; lesser artists literally solve the same problems by successive approximation. One of the reasons kids give up drawing at ten or so is that they decide to start drawing like grownups, and one of the first things they try is a line drawing of a face. Smack! 実際、線画は限りない完璧さを要求するために、視覚的な媒体として最も難しいものだ。数学で言えば、それは閉じた形の解にあたる。劣った芸術家は、実際、同じ線画という問題を何度も近似を繰り返すことにより解いているわけだから。子供が 10 歳くらいで絵を描くことをあきらめてしまう理由のひとつは、大人がやるように描こうとする上に、最初に試すことのひとつが顔の線画だったりするからだ。やれやれ。

In most fields the appearance of ease seems to come with practice. Perhaps what practice does is train your unconscious mind to handle tasks that used to require conscious thought. In some cases you literally train your body. An expert pianist can play notes faster than the brain can send signals to his hand. Likewise an artist, after a while, can make visual perception flow in through his eye and out through his hand as automatically as someone tapping his foot to a beat.

多くの分野で、一見簡単そうなことは練習によってもたらされる。練習の効用とは、意識しなければできなかった事柄を無意識の中でできるようにすることなんだろう。実際に身体を鍛えなければならぬ場合もある。熟練したピアニストは、頭から手に信号が送られるよりも速く音符を弾くことができる。同じように絵描きも、練習の後には、人が足で自然にリズムを取るように視覚の認識が目からそのまま手に流れるようになるのだろう。

When people talk about being in "the zone," I think what they mean is that the spinal cord has the situation under control. Your spinal cord is less hesitant, and it frees conscious thought for the hard problems.

いわゆる「没頭しているとき」というのは、脊髄が全てを制御下に置いているときだと思う。脊髄はためらわないし、意識をより難しい問題に使えるように解放してくれる。

Good design uses symmetry. I think symmetry may just be one way to achieve simplicity, but it's important enough to be mentioned on its own. Nature uses it a lot, which is a good sign.

良いデザインは対称性を使う。対称性は簡潔さに到達する方法のひとつだと思うが、ここで別項目として挙げておくに足る重要性があると思う。自然は対称性をよく使っていて、それは良い指標だ。

There are two kinds of symmetry, repetition and recursion. Recursion means repetition in subelements, like the pattern of veins in a leaf.

対称性には 2 種類ある。繰り返しと再帰だ。再帰とは、葉脈の模様のような、部分構造への繰り返しと言ってもよい。

Symmetry is unfashionable in some fields now, in reaction to excesses in the past. Architects started consciously making buildings asymmetric in Victorian times and by the 1920s asymmetry was an explicit premise of modernist architecture. Even these buildings only tended to be asymmetric about major axes, though; there were hundreds of minor symmetries.

いくつかの分野では、過去に濫用されたために、今では対称性は流行らないものとされている。建築家はビクトリア朝時代に意識的に非対称な建築を始め、1920 年代には非対称性はモダニズム建築の前提とされた。それでも、これらの建築物は主要な軸に対してのみ非対称であることが多い。その中にはたくさんの小さな対称性がある。

In writing you find symmetry at every level, from the phrases in a sentence to the plot of a novel. You find the same in music and art. Mosaics (and some Cezannes) get extra visual punch by making the whole picture out of the same atoms. Compositional symmetry yields some of the most memorable paintings, especially when two halves react to one another, as in the [*Creation of Adam*](#) or [*American Gothic*](#).

文章の中にも、文中の句から小説の筋書きに至るまでさまざまな階層で、対称性を見て取ることが出来る。音楽や美術でもそうだ。モザイク画や、セザンヌの作品のいくつかは、限られた基本要素から全体の画を組み立てることで視覚的なインパクトを高めている。対称的なコンポジションは、特に 2 つの部分が互いに反応している場合、最も印象に残る絵画となる。例えば『アダムとイヴ』や、『アメリカン・ゴシック』のような作品だ。



SYMMETRIC COMPOSITION

In math and engineering, recursion, especially, is a big win. Inductive proofs are wonderfully short. In software, a problem that can be solved by recursion is nearly always best solved that way. The Eiffel Tower looks striking partly because it is a recursive solution, a tower on a tower.

数学や工学では、再帰が特に有用だ。帰納法による証明は素晴らしく短い。ソフトウェアでは、再帰的で解ける問題はほとんどの場合、それが最良の解法だ。エッフェル塔が印象的なのは、おそらくその塔の上に塔がある再帰的な構造にその一部を負っている。

The danger of symmetry, and repetition especially, is that it can be used as a substitute for thought.

対称性、特に繰り返しの危険なところは、それが深く考える代わりに使われてしまい得るという点だ。

Good design resembles nature. It's not so much that resembling nature is intrinsically good as that nature has had a long time to work on the problem. It's a good sign when your answer resembles nature's.

良いデザインは自然に似る。自然に似ているからとにかく良い、というのではなく、自然は問題を解くのに非常に長い時間をかけてきたから、参考にする価値がある。あなたの解が自然の解に似ていたら、それは良いしるしだ。

It's not cheating to copy. Few would deny that a story should be like life. Working from life is a valuable tool in painting too, though its role has often been misunderstood. The aim is not simply to make a record. The point of painting from life is that it gives your mind something to chew on: when your eyes are looking at something, your hand will do more interesting work.

模倣することは悪いことではない。物語は人生のようであるべきだということに反対する

人はほとんどいない。絵画でも、生活から始めるのは良い方法だが、その役割はよく誤解される。単に記録を取るのが目的なのではない。生活の中から素材をとるポイントは、精神で咀嚼するものが自然に得られることだ。素材を見ている間に、手がより面白い仕事をしてくれる。

Imitating nature also works in engineering. Boats have long had spines and ribs like an animal's ribcage. In some cases we may have to wait for better technology: early aircraft designers were mistaken to design aircraft that looked like birds, because they didn't have materials or power sources light enough (the Wrights' engine weighed 152 lbs. and generated only 12 hp.) or control systems sophisticated enough for machines that flew like birds, but I could imagine little unmanned reconnaissance planes flying like birds in fifty years.

工学でも自然を模倣することができる。船は昔から動物の胸部のように、肋骨と背骨を備えていた。ただ、より良い技術を待たねばならない場合もある。初期の飛行機の設計者は飛行機を鳥に似せようとしていた。十分に軽い素材や動力もなく(ライト兄弟のエンジンは152 ポンドで、ほんの12 馬力しか出なかった)、鳥のように飛ばせるだけの制御機構もない時代には、そのような設計は間違いであった。だが私は、これから50年のうちに、鳥のように飛ぶ小さな無人の偵察機が作られるのではないかと想像する。

Now that we have enough computer power, we can imitate nature's method as well as its results. Genetic algorithms may let us create things too complex to design in the ordinary sense.

十分な計算力を手にした現在、私たちは自然の結果だけでなくその手法までも模倣することができる。遺伝的アルゴリズムは、通常的手法では複雑すぎる設計をも行うことができるだろう。

Good design is redesign. It's rare to get things right the first time. Experts expect to throw away some early work. They plan for plans to change.

良いデザインは再デザインだ。最初からうまくできるということは滅多にない。熟練者は初期の仕事は捨てるつもりでやる。計画が変更されることを予定してかかるのだ。

It takes confidence to throw work away. You have to be able to think, *there's more where that came from*. When people first start drawing, for example, they're often reluctant to redo parts that aren't right; they feel they've been lucky to get

that far, and if they try to redo something, it will turn out worse. Instead they convince themselves that the drawing is not that bad, really-- in fact, maybe they meant it to look that way.

仕事を捨てるのには自信が必要だ。「ここまでできたのなら、もっとできるはずだ」こう考えることができなくちゃならない。例えば絵を描き始めたばかりの人は、うまく描けなかった部分をやり直したがる。ここまでできたのはラッキーだった、これ以上何かやったら悪くしてしまうかもしれない、そうってしまうのだ。やり直す代わりに、そんなに悪くないじゃないか、きっとこう描くのが良かったんだと、人は自分を納得させてしまう。

Dangerous territory, that; if anything you should cultivate dissatisfaction. In Leonardo's [drawings](#) there are often five or six attempts to get a line right. The distinctive back of the Porsche 911 only appeared in the redesign of an awkward [prototype](#). In Wright's early plans for the [Guggenheim](#), the right half was a ziggurat; he inverted it to get the present shape.

危険な領域だ。むしろ、不満足な点を突き詰めなければ。ダ・ヴィンチの絵画には、しばしば5回も6回も線を正そうとした跡が見て取れる。ポルシェ 911 の特徴的な背面は、だいたい試作品があって初めて生まれた。ライトのグッゲンハイム美術館の初期の計画では、右半分はジグラット形式になるはずであった。彼は後にそれを現在の形へと改めた。



SKETCH OF A REARING HORSE

Leonardo da Vinci, red chalk, 15.3 x 14.2 cm, c. 1483-1498.

[Click to see a larger version.](#)



PORSCHE 695

1961 prototype of what eventually became the Porsche [911](#).



GUGGENHEIM MUSEUM OF MODERN ART
Frank Lloyd Wright, 1943-1956.

Mistakes are natural. Instead of treating them as disasters, make them easy to acknowledge and easy to fix. Leonardo more or less invented the sketch, as a way to make drawing bear a greater weight of exploration. Open-source software has fewer bugs because it admits the possibility of bugs.

間違ふのは自然なことだ。間違いを大失敗のように考えるのではなく、簡単に見つけて簡単に直せるようにしておくことだ。ダ・ヴィンチは絵画においてより多くの可能性を試すことができるようにスケッチを発明したと言っても、当たらずとも遠からずであろう。オープンソースソフトウェアはバグの可能性を認めているがゆえにバグが少ない。

It helps to have a medium that makes change easy. When oil paint replaced tempera in the fifteenth century, it helped painters to deal with difficult subjects like the human figure because, unlike tempera, oil can be blended and overpainted.

変更を簡単にしておく媒体を持つことは助けになる。15世紀に油絵がテンペラを置き換えて、画家たちは人物画のような難しい対象に取り組みやすくなった。テンペラと違って、油絵具は混ぜたり塗り重ねたりできるからだ。

Good design can copy. Attitudes to copying often make a round trip. A novice imitates without knowing it; next he tries consciously to be original; finally, he decides it's more important to be right than original.

良いデザインは模倣する。模倣に対する態度はしばしば一巡する。初心者は知らず知らずのうちに模倣する。そのうち、彼は意識的に独自性を出そうとする。最後に、独自性よりも正しくあることがより重要だと気付く。

Unknowing imitation is almost a recipe for bad design. If you don't know where your ideas are coming from, you're probably imitating an imitator. Raphael so pervaded mid-nineteenth century taste that almost anyone who tried to draw was imitating him, often at several removes. It was this, more than Raphael's own work, that bothered the Pre-Raphaelites.

気付かずに模倣することは、ほぼ間違いなく悪いデザインをもたらす。どこから自分のアイデアが来たのか知らない場合、たぶんあなたは模倣者の模倣をしている。ラファエルは19世紀中ごろの美術界を席卷したため、その時代に画家を志した者のほとんどは彼を模倣していた。ラファエル前派が反発したのはラファエル自身の作品よりも、そういった風潮であった。

The ambitious are not content to imitate. The second phase in the growth of taste is a conscious attempt at originality.

志ある人は単なる模倣では満足しない。センスの成長の次の段階では意識的に独自性を出そうとする。

I think the greatest masters go on to achieve a kind of selflessness. They just want to get the right answer, and if part of the right answer has already been discovered by someone else, that's no reason not to use it. They're confident enough to take from anyone without feeling that their own vision will be lost in the process.

最も偉大な作り手たちは、ある種の滅私状態に達するのではないかと私は思う。彼らは正しい答えを知りたいだけなのだ。そしてもし、答えの一部が誰かによって発見されていたのなら、それを利用しない手はない。誰かの仕事を借りても自分のビジョンは曇らないという十分な自信があるのだ。

Good design is often strange. Some of the very best work has an uncanny

quality: [Euler's Formula](#), Bruegel's [Hunters in the Snow](#), the [SR-71](#), [Lisp](#). They're not just beautiful, but strangely beautiful.

良いデザインはしばしば奇妙だ。最も優れた仕事のいくつかは、妖しい性質を帯びている。オイラーの公式、ブリューゲルの『雪中の狩人』、ロッキード SR-71、Lisp。これらは単に美しいだけでなく、奇妙な美しさを持っている。



HUNTERS IN THE SNOW
Click to see a larger version.



I'm not sure why. It may just be my own stupidity. A can-opener must seem miraculous to a dog. Maybe if I were smart enough it would seem the most natural thing in the world that $e^{i\pi} = -1$. It is after all necessarily true.

なぜだか私には分からない。単に私が鈍いだけなのかもしれない。缶切りは犬にとって奇跡に見えるだろう。私がもっと賢かったら、 $e^{i\pi} = -1$ であるような世界が一番自然に思えるのかもしれない。結局のところ、それが真理なのだから。

Most of the qualities I've mentioned are things that can be cultivated, but I don't think it works to cultivate strangeness. The best you can do is not squash it if it starts to appear. Einstein didn't try to make relativity strange. He tried to make it true, and the truth turned out to be strange.

私がここで挙げた良いデザインの性質の多くは意識して作り出してゆくことができるが、この奇妙さに関してだけはそうはいかないと思う。せいぜいできるのは、奇妙さが現れてきたときにそれを潰さないように気を付けることくらいだ。アインシュタインは相対性理論をわざわざ奇妙にしようとしたわけではなかった。ただ正しくしようとして、その結果奇妙な真実が現れたのだ。

At an art school where I once studied, the students wanted most of all to develop a personal style. But if you just try to make good things, you'll inevitably do it in a distinctive way, just as each person walks in a distinctive way. Michelangelo was not trying to paint like Michelangelo. He was just trying to paint well; he couldn't help painting like Michelangelo.

私がアートスクールで学んでいた時、学生は何を置いても自分のスタイルを確立していた。だが、ただ良いものを作ろうとすれば、必然的にそれを他とは違ったやり方であるようになるのだ。ちょうど歩き方が人それぞれであるのと同じように。ミケランジェロは、ミケランジェロのように描こうとしたわけではなかった。ただうまく描こうとして、その結果どうしてもミケランジェロのように描いてしまったのである。

The only style worth having is the one you can't help. And this is especially true for strangeness. There is no shortcut to it. The Northwest Passage that the Mannerists, the Romantics, and two generations of American high school students have searched for does not seem to exist. The only way to get there is to go through good and come out the other side.

唯一持つべきスタイルは、どうしてもそうになってしまう、というようなものだ。特に奇妙さにおいてはそうだ。そこに至る近道はない。マニエリスト、ロマン派、そして 2 世代にわたるアメリカの高校生たちが探した北西航路は存在しないようだ。そこに至る唯一の方法は、良いものをひたすら求めて、気が付くと反対側に出ていた、というようなものであり得ない。

Good design happens in chunks. The inhabitants of fifteenth century Florence included Brunelleschi, Ghiberti, Donatello, Masaccio, Filippo Lippi, Fra Angelico, Verrocchio, Botticelli, Leonardo, and Michelangelo. Milan at the time was as big as Florence. How many fifteenth century Milanese artists can you name?

良いデザインは集団で生起する。15 世紀のフィレンツェには、ブルネレスキ、ギベルティ、ドナテルロ、マザッチョ、フィリッポ・リッピ、フラ・アンジェリコ、ヴェロッキオ、ボッティチェリ、レオナルド・ダ・ヴィンチ、そしてミケランジェロがいた。その当時、ミラノはフィレンツェとほぼ同規模の都市だった。あなたは何人のミラノ出身のアーティストを挙げることができるだろうか。

Something was happening in Florence in the fifteenth century. And it can't have been heredity, because it isn't happening now. You have to assume that whatever inborn ability Leonardo and Michelangelo had, there were people born in Milan with just as much. What happened to the Milanese Leonardo?

15 世紀、フィレンツェには何かが起こっていたのだ。それは現在に続いてはいないから、血統のせいではないと言える。レオナルド・ダ・ヴィンチやミケランジェロに何らかの生まれながらの能力が備わっていたとしても、ミラノにも同様の能力を持った人はいたはずだ。ではミラノのレオナルドには何が起こったのだろう。

There are roughly a thousand times as many people alive in the US right now as lived in Florence during the fifteenth century. A thousand Leonardos and a thousand Michelangelos walk among us. If DNA ruled, we should be greeted daily by artistic marvels. We aren't, and the reason is that to make Leonardo you need more than his innate ability. You also need Florence in 1450.

現在、米国には 15 世紀のフィレンツェの人口の約 1000 倍の人々が住んでいる。1000 人のレオナルド、1000 人のミケランジェロが私たちのなかにいるはずだ。DNA が全てを支配するなら、今日私たちは毎日のように素晴らしい芸術に驚嘆しているはずではないか。現実はどうではない。レオナルドがレオナルドになるには、生まれながらの能力以上の何

かなが重要なのだ。1450 年のフィレンツェが重要なのだ。

Nothing is more powerful than a community of talented people working on related problems. Genes count for little by comparison: being a genetic Leonardo was not enough to compensate for having been born near Milan instead of Florence. Today we move around more, but great work still comes disproportionately from a few hotspots: the Bauhaus, the Manhattan Project, the *New Yorker*, Lockheed's Skunk Works, Xerox Parc.

関連した問題を解こうとしている才能ある人々のコミュニティほどパワフルなものはない。それに比較すれば遺伝子の影響なんて小さなものだ。遺伝子的にレオナルドであるだけでは、フィレンツェではなくミラノに生まれてしまった不利を打ち消すことはできない。現代では人々の移動は激しいが、良い仕事は依然としてごく少数の「ホットスポット」から集中して出てくる。バウハウス、マンハッタンプロジェクト、『ニューヨーカー』、ロッキードのスカンクワークス、そしてゼロックスの Parc。

At any given time there are a few hot topics and a few groups doing great work on them, and it's nearly impossible to do good work yourself if you're too far removed from one of these centers. You can push or pull these trends to some extent, but you can't break away from them. (Maybe *you* can, but the Milanese Leonardo couldn't.)

どんな時代にも、少数のホットなトピックがあり、それに向けて多くの仕事をなす少数の集団がいる。それらの中心から遠く離れてしまっていては、良い仕事をするのはほとんど不可能だ。このような流れを押したり引いたりすることはある程度できるが、完全に逃れることは出来ない(いや、あなたならできるかもしれないけど。でもミラノのレオナルドにはできなかった)。

Good design is often daring. At every period of history, people have believed things that were just ridiculous, and believed them so strongly that you risked ostracism or even violence by saying otherwise.

良いデザインはしばしば大胆だ。歴史のどの時代にも、馬鹿げたことがあまりに人々に固く信じられていて、それと違うことを口にしようものなら追放されたり、ときに暴力を振るわれることさえあった。

If our own time were any different, that would be remarkable. As far as I can tell

it [isn't](#).

私たちの時代は違うというのならそれは素晴らしいことだ。だが私の見る限り、私たちの時代も同じだ。

This problem afflicts not just every era, but in some degree every field. Much Renaissance art was in its time considered shockingly secular: according to Vasari, Botticelli repented and gave up painting, and Fra Bartolommeo and Lorenzo di Credi actually burned some of their work. Einstein's theory of relativity offended many contemporary physicists, and was not fully accepted for decades-- in France, not until the 1950s.

この問題は全ての時代だけでなく全ての分野に影を投げかけてきた。ルネッサンスの芸術の多くは当時はあまりに世俗的だと考えられた。ヴァサリによれば、ボッティチェリは悔やんで筆を折り、フラ・バルトロメオとロレンツォ・ディ・クレディは自分の作品のいくつかを実際に燃やしてしまったという。アインシュタインの相対性理論に多くの当時の物理学者は拒否反応を示したし、その後長い間(とりわけフランスでは、1950年代になるまで)完全には受け入れられなかった。

Today's experimental error is tomorrow's new theory. If you want to discover great new things, then instead of turning a blind eye to the places where conventional wisdom and truth don't quite meet, you should pay particular attention to them.

今日の実験の誤りは明日の新理論だ。新しいことを発見したければ、主流の知識と真実が一致しないところに目をつむるのではなく、逆にそこに注目しなければならない。

As a practical matter, I think it's easier to see ugliness than to imagine beauty. Most of the people who've made beautiful things seem to have done it by fixing something that they thought ugly. Great work usually seems to happen because someone sees something and thinks, *I could do better than that*. Giotto saw traditional Byzantine madonnas painted according to a formula that had satisfied everyone for centuries, and to him they looked wooden and unnatural. Copernicus was so troubled by a hack that all his contemporaries could tolerate that he felt there must be a better solution.

実際問題としては、美を想像するよりは醜に目をやるほうが簡単だと思う。美しいものを作った人々の多くは、彼らが醜いと思ったところを直していったのだと思える。誰かが何

かを見てこう考える、「俺ならもっとうまくできる」と。これが偉大な仕事の発端なのだろう。ジョットは人々が何世紀にもわたって満足してきた方式で描かれたビザンチンのマドンナ像を見て、無表情で不自然だと感じた。コペルニクスは、同時代の他の学者が許容してきたツギハギにどうしても我慢できず、もっと良い方法があるに違いないと感じたのだ。

Intolerance for ugliness is not in itself enough. You have to understand a field well before you develop a good nose for what needs fixing. You have to do your homework. But as you become expert in a field, you'll start to hear little voices saying, *What a hack! There must be a better way.* Don't ignore those voices. Cultivate them. The recipe for great work is: very exacting taste, plus the ability to gratify it.

醜いものを許せないだけでは十分ではない。どこを直せばよいのか知る嗅覚を得るためには、その分野を十分に理解していなければならない。しっかり勉強しなくちゃならないんだ。だがその分野で熟練者となれば、内なる声が囁き出すだろう。「なんてハックだ！もっと良い方法があるはずだ」この声を無視してはいけない。それを追求するんだ。厳しい味覚と、それを満足させる能力。それが偉大な仕事のためのレシピだ。

Notes

注釈

[Sullivan](#) actually said "form ever follows function," but I think the usual misquotation is closer to what modernist architects meant.

サリヴァンは実は「形態は常に機能にしたがう(form ever follows function)」と言った。だがよく言われる引用違いのほうがモダニズム建築家の意味しているものに近いように思える。

Stephen G. Brush, "Why was Relativity Accepted?" *Phys. Perspect.* 1 (1999) 184-214.

Chapter 10 PLOGRAMMING LANGUAGES EXPLAINED

第 10 章 プログラミング言語入門

Any machine has a list of things you can tell it to do. Sometimes the list is short. There are only two things I can do to my electronic kettle: turn it on and turn it off. My CD player is more complicated. As well as turning it on and off, I can turn the volume up and down, tell it to play or pause, move back or forward one song, and ask it to play songs in random order.

どんな機械も、あなたがそれに対して命令できることのリストを持っている。リストはごく短い場合もある。例えば電気ポットには 2 つのことしか命令できない。オンにするか、オフにするかだ。CD プレーヤーはもう少し複雑だ。オンとオフの他に、ボリュームを上げたり下げたり、再生したり一時停止したり、1 曲分進んだり戻ったり、またランダムに再生するように指示することだってできる。

Like any other kind of machine, a computer has a list of things it can do. For example, every computer can be told to add two numbers. The complete list of things a computer can do is its machine language.

他のいろいろな機会と同じように、コンピュータもそれができることのリストを持っている。例えば、どんなコンピュータも 2 つの数値を足すことができる。コンピュータができる全てのことをまとめたものは、機械語と呼ばれる。

Machine Language

機械語

When computers were first invented, all programs had to be written as sequences of machine language instructions. Soon after, they started to be written in a slightly more convenient form called assembly language . In assembly language the list of commands is the same, but you get to use more programmer-friendly names. Instead of referring to the add instruction as 11001101, which is what the machine might call it, you get to say add.

コンピュータが最初に発明された時、全てのプログラムは機械語の命令列としてかかれなければならなかった。じきに、それはもう少し便利な形式であるアセンブリ言語として書かれるようになった。アセンブリ言語では、使える命令の種類は同じだが、プログラマに分かりやすい名前が使えるようになっている。例えば足し算の命令を、機械が理解する 11001101 というような形式ではなく、add のような名前で参照するんだ。

The problem with machine/assembly language is that most computers can only do very simple things. For example, suppose you want to tell a computer to beep 10 times. There's not likely to be a machine instruction to do something n times. So if you wanted to tell a computer to do something 10 times using actual machine instructions, you'd have to say something equivalent to:

機械語やアセンブリ言語の問題点は、ほとんどのコンピュータはものすごく単純な動作しかできないということだ。例えば、コンピュータに 10 回ブザーを鳴らすように命令したいとしよう。「何かを n 回繰り返す」というような命令をコンピュータが持っていることはあまりない。したがって、コンピュータに何かを 10 回やるように機械語で命令したければ、こんな感じの命令を書かなければならない。

```

    put the number 10 in memory location 0
a    if location 0 is negative, go to line b
    beep
    subtract 1 from the number in location 0
    go to line a
b    ...rest of program...
    メモリの番地 0 に数値 10 を格納せよ
a    もし番地 0 の数値が負ならば、b へ飛べ
    ブザーを鳴らせ
    番地 0 の数値から 1 を減算せよ
    a へ飛べ
b    ...プログラムの続き...
```

If you have to do this much work to make the machine beep 10 times, imagine the labor of writing something like a word processor or a spreadsheet.

たかだか、ブザーを 10 回鳴らすだけでこれだけのことを書かなくちゃならないとしたら、ワープロや表計算ソフトを書くのにどれだけ手間がかかるか、想像に難くないだろう。

And by the way, take another look at the program. Will it actually beep ten times? Nope, eleven. In the first line I should have said 9 instead of 10. I deliberately put a bug in our example to illustrate an important point about languages. The more you have to say to get something done, the harder it is to see bugs.

しかもだ。先に挙げたプログラムをよく見てほしい。ブザーは 10 回鳴るだろうか。いや、11 回だ。最初の行で、私は 10 ではなく 9 と書かなければならなかった。ここで私は、言語についての重要な点を見せるために、わざと例にバグを忍び込ませてみたんだ。何かを

するための命令が多くなるほど、バグを見つけるのは難しくなる。

High-Level Languages

高級言語

Imagine you had to produce assembly language programs, but you had an assistant to do all the dirty work for you. So you could just write something like アセンブリ言語でプログラムを書くときに、面倒なことを全部やってくれる助手がいたとしたらどうだろう。例えば、あなたが

`dotimes 10 beep`

繰り返し 10 回ブザーを鳴らせ

and your assistant would write the assembly language for you (but without bugs). と書いたとすると、その助手が、バグのないアセンブリ言語を書いてくれるんだ。

In fact, this is how most programmers do work. Except the assistant isn't a person, but a compiler. A compiler is a program that translates programs written in a convenient form, like the one liner above, into the simple-minded language that the hardware understands.

実は、ほとんどのプログラマはそんなふうに仕事をしている。ただし、この助手は人間ではなく、コンパイラだ。コンパイラは、上に挙げた 1 行のような分かりやすい形式で書かれたプログラムを、ハードウェアが理解できる単純な言語へと変換してくれるプログラムだ。

The more convenient language that you feed to the compiler is called a highlevel language. It lets you build your programs out of powerful commands, like "do something n times" instead of wimpy ones like "add two numbers."

コンパイラに与える、より便利な形の言語は、高級言語と呼ばれる。それを使えば、「2 つの数を足せ」みたいな弱っちい命令だけじゃなくて、「何かを n 回繰り返せ」みたいな強力な命令を使ってプログラムを組み立てることができるようになるんだ。

When you get to build your programs out of bigger concepts, you don't need to use as many of them. Written in our imaginary high-level language, our program is only a fifth as long. And if there were a mistake in it, it would be easy to see. 大きな概念からプログラムを組み立てれば、たくさんの命令を費やさなくても済む。上に

例を挙げた仮想的な高級言語では、プログラムは 5 分の 1 の長さになった。そして、間違いがあったとしても、それはずっと見つけやすいだろう。

Another advantage of high-level languages is that they make your programs more portable. Different computers all have slightly different machine languages. You cannot, as a rule, take a machine language program written for one computer and run it on another. If you wrote your programs in machine language, you'd have to rewrite them all to run them on a new computer. If you use a high-level language, all you have to rewrite is the compiler.

高級言語のもうひとつの利点は、プログラムがより移植しやすくなるということだ。異なるコンピュータは、少しずつ違う機械語を使っている。原則として、あるコンピュータ向けに書かれた機械語のプログラムをそのまま他の種類のコンピュータで走らせることはできない。プログラムが機械語で書かれていたら、新しいコンピュータでそれを走らせるためには全て書き直さないとならないんだ。高級言語を使っていれば、コンパイラだけを書き直せば済む。

Compilers aren't the only way to implement high-level languages. You could also use an interpreter, which examines your program one piece at a time and executes the corresponding machine language commands, instead of translating the whole thing into machine language and running that.

高級言語を実現する方法はコンパイラだけではない。インタプリタというものを使う方法もある。与えられたプログラムを一括して機械語に変換してから実行する代わりに、インタプリタはプログラムを一行ずつ調べて、対応する機械語の命令列を実行してゆくものだ。

Open Source

オープンソース

The high-level language that you feed to the compiler is also known as source code, and the machine language translation it generates is called object code. When you buy commercial software, you usually only get the object code. (Object code is so hard to read that it is effectively encrypted, thus protecting the company's trade secrets.) But lately there is an alternative approach: open source software, where you get the source code as well, and are free to modify it if you want.

コンパイラに与える高級言語で書かれたプログラムはまた、ソースコードとも呼ばれる。それが変換されてできた機械語はオブジェクトコードと呼ばれる。売っているソフトウェ

アを買ってくる場合、普通はオブジェクトコードしか手に入らない(オブジェクトコードは読むのが非常に難しく、実質的には暗号化してあるようなものなので、それを作っている会社の企業秘密は守られる)。しかし近年、別の方法が盛んになってきた。オープンソースソフトウェアと呼ばれるものは、ソースコードも手に入れることができ、必要ならばそれを変更することもできる。

There is a real difference between the two models. Open source gives you a lot more control. When you're using open source software and you want to understand what it's doing, you can read the source code and find out. If you want, you can even change the software and recompile it.

この 2 つのモデルには重要な違いがある。オープンソースは、使い手であるあなたにより大きな力を与える。オープンソースソフトウェアを使っていて、それがどういうふうに動いているかを理解したかったら、ソースコードを読めばよい。そして、やりたければ、ソフトウェアを変更して再コンパイルすることさえ可能だ。

One reason you might want to do that is to fix a bug. You can't fix bugs in Microsoft Windows, for example, because you don't have the source code. (In theory you could hack the object code, but in practice this is very hard. It's also probably forbidden by the license agreement.) This can be a real problem. When a new security hole is discovered in Windows, you have to wait for Microsoft to release a fix. And security holes at least get fixed fast. If the bug merely paralyzes your computer occasionally, you may have to wait till the next full release for it to be fixed.

そうしたい理由のひとつは、バグを修正することだ。例えば、あなたはマイクロソフトの **Windows** のバグを修正することはできない。ソースコードがないからだ(オブジェクトコードをハックすることは理論的には可能だが、現実には非常に難しい。また、その行為はライセンス契約で禁じられているだろう)。この事実は重大な問題になり得る。**Windows** に新しいセキュリティ上の問題が見つかった場合、あなたはマイクロソフトがその問題を修正してくれるまで待たなければならないんだ。セキュリティ上の問題はそれでも早く修正されるほうだ。バグが、コンピュータを時々固まらせる程度の問題ならば、その修正を次の大きなリリースまで待たされることになりかねない。

But the advantage of open source isn't just that you can fix it when you need to. It's that everyone can. Open source software is like a paper that has been subject to peer review. Lots of smart people have examined the source code of open source operating systems like Linux and FreeBSD and have already found most of

the bugs. Whereas Windows is only as reliable as big-company QA can make it. オープンソースの利点は、必要ならばあなたがバグを修正できるというだけじゃない。誰もがそうできるというところにあるんだ。オープンソースソフトウェアは、専門家同士で審査される論文のようなものだ。たくさんの賢い人々が、Linux や FreeBSD といったオープンソースのオペレーティングシステムのソースコードを調べて、既にたくさんのバグを見つけている。Windows では、大企業の品質管理部門の信頼性に頼るしかないというのに。

Open source advocates are sometimes seen as wackos who are against the idea of property in general. A few are. But I'm certainly not against the idea of property, and yet I would be very reluctant to install software I didn't have the source code for. The average end user may not need the source code of their word processor, but when you really need reliability, there are solid engineering reasons for insisting on open source.

オープンソースを熱狂的に支持する人々は、しばしば私有財産一般に対して反対する変人のように思われることがある。確かにそういう人もいる。だが、私自身は財産というものに対して反対はしていない。それでも私は、自分がソースコードを持っていないソフトをコンピュータに入れるのには非常に消極的だ。普通のエンドユーザはワープロのソースコードなんかなくてもよいだろうが、本当に信頼性が必要となる場所では、オープンソースを強く主張する、工学的に確固とした理由があるのだ。

Language Wars

言語戦争

Most programmers, most of the time, program in high-level languages. Few use assembly language now. Computer time has become much cheaper, while programmer time is as expensive as ever, so it's rarely worth the trouble of writing programs in assembly language. You might do it in a few critical parts of, say, a computer game, where you wanted to micromanage the hardware to squeeze out that last increment of speed.

ほとんどのプログラマは、ほとんどいつでも、高級言語でプログラムする。アセンブリ言語を使う人は少数派だ。計算機の時間はずっと安価になり、一方でプログラマの時間は高価なのだから、アセンブリ言語でプログラムを書くことに時間を費やす意味があることは滅多にない。もちろん、ハードウェアに可能なスピードの最後の一滴までを絞り取る必要があるような、例えばコンピュータゲームの重要な部分などでは、アセンブリ言語が必要になる場面もあるにせよ。

Fortran, Lisp, Cobol, Basic, C, Pascal, Smalltalk, C++, Java, Perl, and Python are all high-level languages. Those are just some of the better known ones. There are literally hundreds of different high-level languages. And unlike machine languages, which all offer similar instruction sets, these high-level languages give you quite different concepts to build programs out of.

Fortran、Lisp、Cobol、Basic、C、Pascal、Smalltalk、C++、Java、Perl、Python、これらは全て高級言語だ。ここに挙げたのは比較的知られているものだけで、この他にも何百種類もの高級言語がある。機械語はだいたいどれも似たような命令しか持っていないが、これらの高級言語は、プログラムを構築するための概念として非常に異なるものをそれぞれ提供している。

So which one do you use? Ah, well, there is a great deal of disagreement about that. Part of the problem is that if you use a language for long enough, you start to think in it. So any language that's substantially different feels terribly awkward, even if there's nothing intrinsically wrong with it. Inexperienced programmers' judgements about the relative merits of programming languages are often skewed by this effect.

では、どれを使えばいいんだろう。いやはや、それは尽きぬ論争の種だ。問題の一部は、ある言語を十分に長く使っていると、その言語で考えるようになるということだ。だから自分の使っている言語と大きく異なるものは、たとえ何も悪いことがなかったとしても、ものすごく不格好に感じられる。経験の浅いプログラマによる言語の比較は、しばしばこの効果によって歪められている。

Other hackers, perhaps from a desire to seem sophisticated, will tell you that all languages are basically the same. I've programmed in all kinds of languages, said the tough old hacker as he eased up to the bar, and it don't matter which you use. What matters is whether you have the right stuff. Or something along those lines. 他のハッカーは、たぶん格好つけたいからだろうが、どの言語も基本的には同じだよ、と教えたがるかもしれない。「わしは実にさまざまな種類の言語でプログラムを書いたが、どれを使うかはあまり関係がないな。素質を持っているかどうかの問題じゃ」なんてことを、バーで寛いでいる歴戦の老ハッカーは語るかもしれない。

This is nonsense, of course. There is a world of difference between, say, Fortran I and the latest version of Perl—or for that matter between early versions of Perl and the latest version of Perl. But the tough old hacker may himself believe what he's saying. It's possible to write the same primitive Pascal-like programs in

almost every language. If you only ever eat at McDonald's, it will seem that food is much the same in every country.

もちろん、こんなことはナンセンスだ。例えば、Fortran I と最新の Perl の間には、あるいは、それをいうなら初期の Perl と最新の Perl の間にも、天と地ほどの差がある。でも、例の歴戦の老ハッカーは自分の言っていることを信じているかもしれない。Pascal 的なプログラムを書くための基本構文は、ほとんどどの言語にもあるからだ。あなたがどこへ行ってもマクドナルドで食事するなら、食事はどんな国でも同じようなものだという結論に達するだろう。

Some hackers prefer the language they're used to, and dislike anything else. Others say that all languages are the same. The truth is somewhere between these two extremes. Languages do differ, but it's hard to say for certain which are best. The field is still evolving.

ハッカーの一部は、自分の慣れた言語を好み、他の全てを嫌う。他のハッカーの一部は、全ての言語は同じだと言う。真実はこの両極端の間のどこかにある。言語には確かに違いがあるが、どれが一番かを確実に判断するのは難しい。この分野はまだ進化途上にある。

Abstractness

抽象度

Just as high-level languages are more abstract than assembly language, some high-level languages are more abstract than others. For example, C is quite lowlevel, almost a portable assembly language, whereas Lisp is very high-level. 高級言語はアセンブリ言語よりも抽象度が高い。同様に、ある種の高級言語は他的高级言語よりも抽象的だ。例えば、C 言語は、ポータブルなアセンブリ言語と言ってもよいくらいに低レベルだが、Lisp は非常に高レベルだ。

If high-level languages are better to program in than assembly language, then you might expect that the higher-level the language, the better. Ordinarily, yes, but not always. A language can be very abstract, but offer the wrong abstractions. I think this happens in Prolog, for example. It has fabulously powerful abstractions for solving about 2% of problems, and the rest of the time you're bending over backward to misuse these abstractions to write de facto Pascal programs.

高級言語がアセンブリ言語よりもプログラムするのによいならば、より高級な言語のほうがより良いと考えるのは自然だろう。たいていはそうだが、いつもそうではない。言語は

非常に抽象的であり得るが、適切でない抽象化がなされている場合もある。例えば、**Prolog** がそのケースだと私は思う。**Prolog** は非常に強力な抽象化を提供するが、それで解けるのは問題の 2% くらいの部分だ。残りの部分はよくある **Pascal** 風のプログラムを書くために、その強力な抽象化機構を不自然にねじ曲げなければならない。

Another reason you might want to use a lower-level language is efficiency. If you need code to be super fast, it's better to stay close to the machine. Most operating systems are written in C, and it is not a coincidence. As hardware gets faster, there is less pressure to write applications in languages as low-level as C, but everyone still seems to want operating systems to be as fast as possible. (Or maybe they want the prospect of buffer-overflow attacks to keep them on their toes.)

低レベルの言語を使いたいもうひとつの理由は効率だ。ものすごく高速に動作するコードが欲しければ、なるべく機械に近いところにいるのがいい。多くのオペレーティングシステムは **C** で書かれているが、それは偶然ではない。ハードウェアが早くなるにしたがって、アプリケーションを **C** のような低レベルの言語で書かなければならないという圧力は少なくなってきたが、オペレーティングシステムはまだ可能な限り速くなくてはならないと思われるようだ(あるいは、自分たちの気を引き締めるために、バッファオーバーフロー攻撃の危険性が必要なかもしれない)。

Seat Belts or Handcuffs?

シートベルト、それとも手錠?

The biggest debate in language design is probably the one between those who think that a language should prevent programmers from doing stupid things, and those who think programmers should be allowed to do whatever they want. Java is in the former camp, and Perl in the latter. (Not surprisingly, the DoD is big on Java.)

言語の設計において一番の大きな議論は、言語はプログラマの愚かな行いを防ぐべきだと考える一派と、プログラマがやりたいことは何でも許すべきだと考える一派との間にある。**Java** は前者に属し、**Perl** は後者に属している(国防省が **Java** に肩入れしているのもっともなことだ)。

Partisans of permissive languages ridicule the other sort as "B&D" (bondage and discipline) languages, with the rather impudent implication that those who like to program in them are bottoms. I don't know what the other side call languages like

Perl. Perhaps they are not the sort of people to make up amusing names for the opposition.

何でも許される言語を使う職人たちは、相手の言語のことを、そんな言語でプログラムしているような連中は下っ端だ、という生意気な考えをも込めて、「緊縛と調教(bondage and discipline, B&D)」の言語などと呼んで茶化す。相手の側が Perl のような言語を何と呼んでいるか、私は知らない。たぶんあちらは、相手にあだ名を付けて楽しむような種類の連中ではないんだろう。

The debate resolves into several smaller ones, because there are several ways to prevent programmers from doing stupid things. One of the more active questions at the moment is static versus dynamic typing. In a statically-typed language, you have to know the kind of values each variable can have at the time you write the program. With dynamic typing, you can set any variable to any value, whenever you want.

この議論は、いくつかの小さな議論へと分解できる。プログラマの愚かな行動を防ぐにはいくつかやり方があるからだ。現在、一番活発に交わされている議論は、静的型付けか動的型付けかに関するものだ。静的型付け言語では、プログラムを書く時点で、どのような種類の値がどの変数に格納されるかをあらかじめ知っていなければならない。動的型付けでは、どんな変数にも好きな値を格納できる。

Advocates of static typing argue that it helps to prevent bugs and helps compilers to generate fast code (both true). Advocates of dynamic typing argue that static typing restricts what programs you can write (also true). I prefer dynamic typing. I hate a language that tells me what to do. But some smart people seem to like static typing, so the question must still be an open one.

静的型付け派は、それによってバグを防ぎ、またコンパイラがより速いコードを生成できると主張する(これはどちらも正しい)。動的型付け派は、静的型付けは書けるプログラムを制限すると主張する(これも正しい)。私自身は動的型付けが好きだ。どうすべきかを言語に指示されるのは私は嫌いだ。しかし一部のとても賢い人々は動的型付けを好んでいるようだ。この議論はまだ結論が出ていない。

〇〇

オブジェクト指向

Another big topic at the moment is object-oriented programming. It means a different way of organizing programs. Suppose you want to write a program to

find the areas of two-dimensional figures. At first it only has to know about circles and squares. One way to do it would be to write a single piece of code, within which you test whether you're being asked about a circle or a square, and then use the corresponding formula to find the area. The object-oriented way to write this program would be to create two classes, circle and square, and then attach to each class a snippet of code (called a method) for finding the area of that type of figure. When you need to find the area of something, you ask what its class is, retrieve the corresponding method, and run that to get the answer.

もうひとつの大きな関心の的は、オブジェクト指向プログラミングというものだ。これは、プログラムを組み立てていく方法の違いを意味している。例えば、2次元の図形の面積を計算するプログラムを書いているとしよう。当初は、円と正方形だけを扱えばよいと言われたとする。ひとつの方法は、面積を求める計算のコードをひとつ書いて、その中で質問が円に関するものか正方形に関するものかを教えてもらい、適切な公式を使うようにするというものだ。オブジェクト指向的な方法とは、ここで円と正方形というふたつのクラスを作り、それぞれのクラスに各タイプの面積を計算するコード片(メソッドと呼ばれる)をくっつける、というものだ。何かの面積が必要になったら、そのクラスを尋ね、必要なメソッドを見つけて、それを起動すれば答えが得られる。

These two cases may sound very similar, and indeed what actually happens when you run the code is much the same. (Not surprisingly, since you're solving the same problem.) But the code can end up looking quite different. In the object-oriented version, the code for finding the areas of squares and circles may even end up in different files, one part in the file containing all the stuff to do with circles, and the other in the file containing the stuff to do with squares.

この2つの方法にはたいして違いがないように思えるかもしれない。実際、コードを走らせた時に起こっていることには対して違いがない(それも当然だ。同じ問題を解いているのだから)。だが、コードはとても違った見かけになる。オブジェクト指向では、正方形と円それぞれの面積を求めるコードは、全然別のファイルに置かれることになるかもしれない。ひとつのファイルには円に関する全てのことが書かれ、もうひとつのファイルには、正方形に関する全てのことが書かれる。

The advantage of the object-oriented approach is that if you want to change the program to find the area of, say, triangles, you just add another chunk of code for them, and you don't even have to look at the rest. The disadvantage, critics would counter, is that adding things without looking at what was already there tends to produce the same results in programs that it does in buildings.

オブジェクト指向の利点は、例えばプログラムが三角形の面積も扱わなくちゃならなくなった時に、単に三角形を扱うコードを足すだけで済むことだ。他のプログラムの部分を見る必要さえない。一方、批判派が指摘する欠点とは、他を見ないでもコードを足せることから、既に組み込みで可能なことと同じ結果を出すプログラムがたくさん存在することになりがちだということだ。

The debate about object-oriented programming is not as clear-cut as the one about static versus dynamic typing. With typing you have to choose one or the other. But the object-orientedness of a language is a matter of degree. Indeed, there are two senses of object-oriented: some languages are object-oriented in the sense that they let you program in that style, and others in the sense that they force you to.

オブジェクト指向プログラミングに関する議論は、静的型付けと動的型付けの議論のようにはっきりしたものではない。型付けの議論では、どちらかを選ばなければならなかった。しかし、言語のオブジェクト指向機能にはいろいろな段階がある。実際、言語がオブジェクト指向であると言ったとき、そこは大きく 2 つの意味がある。いくつかの言語では、オブジェクト指向とは、それを使ってもプログラミングができるということであり、別の言語では、オブジェクト指向とは、それを使わなければプログラムが書けないことを指す。

I see little advantage in the latter. Surely a language that lets you do x is at least as good as one that forces you to. So as regards languages, at least, we can finesse this question. Sure, use a language that lets you write object-oriented programs. Whether you ever actually want to then becomes a separate question. 私は、後者の言語にはあまり利点を感じない。少なくとも、何かを使うことが許されている言語は、何かを使うことを強制されている言語と同等以上ではあるはずだ。したがって、こと言語に関しては、この質問にはこんなふうに答えられるだろう。そうだね、もちろん、オブジェクト指向でプログラムが書ける言語を使えばいいよ、と。ただ、そうしたいかどうかはまた別の問題だけれどね。

Renaissance

ルネッサンス

One thing I think everyone in the language business will agree on is that there are a lot of new programming languages lately. Until the 1980s, only institutions could afford the hardware needed to develop programming languages, and so most were designed by professors or researchers at large companies. Now a high

school kid can afford all the hardware necessary.

言語ビジネスにいる人々の誰もが同意できることがひとつある。近年になって非常に多くの新しいプログラミング言語が出現してきたということだ。1980年代までは、何らかの組織でないと、プログラミング言語を開発するのに必要なハードウェアを持つことができなかった。だから言語は主として教授や大企業の研究者らによって作られていた。現在では、高校の生徒だって自分で必要な機材を揃えることができる。

Inspired largely by the example of Larry Wall, the designer of Perl, lots of hackers are thinking, why can't I design my own language? Those who manage to harness the power of the open source community can get a lot of code written for them very quickly.

Perl の設計者であるラリー・ウォールの例に倣って、たくさんのハッカーが、僕だって自分の言語が作れるんじゃないかと考えている。そして、オープンソースコミュニティの力を得ることができれば、人々がたくさんのコードをそれを書いてくれるようになるだろう。

The result is a kind of language you might call top-heavy: a language whose inner core is not very well designed, but which has enormously powerful libraries of code for solving specific problems. (Imagine a Yugo with a jet engine bolted to the roof.) For the little, everyday problems that programmers spend so much of their time solving, libraries are probably more important than the core language. And so these odd hybrids are quite useful, and become correspondingly popular. A Yugo with a jet engine bolted to the roof might actually work, as long as you didn't try to take a corner in it.

結果としてできる言語は、頭でっかちの言語になりがちだ。内部の核はあまりよく設計されていないが、特定の問題を解くための強力なライブラリを豊富に備えている(トラバントの屋根にジェットエンジンが取り付けられている姿を想像したまえ)。少なくとも、プログラマが毎日時間を費やして取り組んでいる問題の多くについては、ライブラリが言語の核よりも重要だ。したがってこの奇妙な取り合わせはなかなか使い出があって、相応に広く使われるようになる。ジェットエンジン付きトラバントだって、使えることは使えるかもしれない。それで角を曲がろうとしなければ、だが。

Another result is a great deal of variety. There has always been a lot of variety in programming languages. Fortran, Lisp, and APL differ from one another as much as starfish, bears, and dragonflies, and all were designed before 1970. But the new open source languages have certainly continued this tradition.

もうひとつの結果は、大きな多様性だ。プログラミング言語には常に多様性があった。

Fortran と Lisp と APL は、ヒトデと熊とトンボほどに違う。どれも 1970 年以前に設計されたものだ。新しいオープンソース言語はこの傾向をずっと後押ししているようだ。

I seem to hear about a new language every couple days. Jonathan Erickson has called it "the programming language renaissance." Another phrase people sometimes use is "the language wars." But there is no contradiction here. The Renaissance was full of wars.

2 日に 1 度は、新しい言語について耳にする。ジョナサン・エリクソンはこれを「プログラミング言語のルネッサンス」と呼んだ。他の人々がしばしば使う用語は、「言語戦争」というものだ。ここに矛盾はない。ルネッサンス時代は戦争だらけだったのだから。

Indeed, many historians believe that the wars were a byproduct of the forces that created the Renaissance. The key to Europe's vigor may have been the fact that it was divided up into a number of small, competing states. These were close enough that ideas could travel from one to the other, but independent enough that no one ruler could put a lid on innovation—as the Chinese court disastrously did when they forbade the development of large ocean-going ships.

実のところ、多数の歴史家は、戦争はルネッサンスを作り出した力の副作用だったと信じている。ヨーロッパの活況をもたらしたのは、それが小さな、互いに競争している諸州に分かれていたからだというのだ。各国は、思想の交流が可能な程度に近く、しかし互いに独立していたので、中国の法廷が大洋を航海できる大型船舶の製造を禁じた時のように、唯一の支持者が革新に蓋をしてしまうことは避けられたのだ。

So it is probably all to the good that programmers live in a post-Babel world. If we were all using the same language, it would probably be the wrong one.

だから、このバベルの塔以降の世界にプログラマーが生きているということは良いことだと言えるのかもしれない。もし私たち皆がひとつの言語しか使っていないのだとしたら、おそらくそれは何か間違った言語であったことだろう。

Chapter 11 THE HUNDRET-YEAR LANGUAGE

第 11 章 百年の言語

April 2003



(This essay is derived from a keynote talk at PyCon 2003.)

(この記事は PyCon 2003 で行った基調講演をもとに書いたものである。)

It's hard to predict what life will be like in a hundred years. There are only a few things we can say with certainty. We know that everyone will drive flying cars, that zoning laws will be relaxed to allow buildings hundreds of stories tall, that it will be dark most of the time, and that women will all be trained in the martial arts. Here I want to zoom in on one detail of this picture. What kind of programming language will they use to write the software controlling those flying cars?

100 年後の生活というものを予測するのは難しい仕事だ。確信を持って言えることなどほんの少ししかない。人々は飛行自動車で飛び回り、建築規制が緩和されて数百階建てのビルがそびえ立ち、なぜだかいつも薄暗くて、女性はみな武術を身に付けているだろうとか、そのくらいのもんだ。ここでは、その 100 年後の世界の中でも、あるひとつの点に注目したい。そんな飛行自動車の制御をするソフトウェアを書くのに、人々はどんなプログラミング言語を使うのだろうか？

This is worth thinking about not so much because we'll actually get to use these languages as because, if we're lucky, we'll use languages on the path from this point to that.

将来そういう言語を実際に使えるまで生きてはいられないだろうけれど、こんなことを考えるのは、うまくすれば現在からそこに至るまでの道筋にある言語を使えるかもしれないからだ。

I think that, like species, languages will form evolutionary trees, with dead-ends branching off all over. We can see this happening already. Cobol, for all its sometime popularity, does not seem to have any intellectual descendants. It is an evolutionary dead-end-- a Neanderthal language.

生物種と同様に、言語も枝分かれや行き止まりがそこらじゅうにある進化の系統樹を形成する。既にその一部は現在でも見られる。Cobol はかつてあれだけ流行ったのに、知能ある子孫を残してはいないようだ。それは進化の行き止まり、ネアンデルタール言語なんだ。

I predict a similar fate for Java. People sometimes send me mail saying, "How can you say that Java won't turn out to be a successful language? It's already a successful language." And I admit that it is, if you measure success by shelf space taken up by books on it (particularly individual books on it), or by the number of undergrads who believe they have to learn it to get a job. When I say Java won't turn out to be a successful language, I mean something more specific: that Java will turn out to be an evolutionary dead-end, like Cobol.

Java も似たような運命をたどるのではないだろうか。時々、「Java が成功する言語にならないなんて言うのはどうかしてるよ。もう成功してるじゃないか」というようなメールを私に送ってくる人がいる。確かに、成功の度合いをその言語についての本が本棚を占有する幅(それも個人所有の本)だとか、あるいは就職に必要だと信じてその言語を勉強する大学生の数で測るなら、Java は成功していると言えるだろう。私が Java は成功する言語にならないだろうと言うのは、もう少し限定した意味においてだ。すなわち Java は、Cobol がそうであったように、進化の袋小路に向かうだろうということだ。

This is just a guess. I may be wrong. My point here is not to dis Java, but to raise the issue of evolutionary trees and get people asking, where on the tree is language X? The reason to ask this question isn't just so that our ghosts can say, in a hundred years, I told you so. It's because staying close to the main branches is a useful heuristic for finding languages that will be good to program in now.

もちろんこれは単なる推測だ。間違っているかもしれない。ここで言いたいのは Java をけなすことじゃなくて、言語の進化樹というものがあるということ、そして言語 X はその樹のどこに位置するんだろうということを問うてもらいたいからだ。この質問は、別に 100 年後に私たちの幽霊が「ほらごらん、言った通りだろう」なんていうためじゃない。樹のなるべく主要な枝に近くあることが、現在プログラムを書くのに良い言語を見つける、経験的にうまい方法だからだ。

At any given time, you're probably happiest on the main branches of an

evolutionary tree. Even when there were still plenty of Neanderthals, it must have sucked to be one. The Cro-Magnons would have been constantly coming over and beating you up and stealing your food.

どんな時点でも、進化樹の主要な枝にいることは一番幸せなことだろう。ネアンデルタール人がたくさんいた時だって、ネアンデルタール人であることは決して良いことではなかったに違いない。だっていつもクロマニヨン人がやってきて、ぼこぼこにされて食糧を奪われていたのだから。

The reason I want to know what languages will be like in a hundred years is so that I know what branch of the tree to bet on now.

100 年後のプログラミング言語がどうなっているかを考えたい理由は、それによって現在のどの枝に賭けるべきかを知りたいからだ。

The evolution of languages differs from the evolution of species because branches can converge. The Fortran branch, for example, seems to be merging with the descendants of Algol. In theory this is possible for species too, but it's not likely to have happened to any bigger than a cell.

言語の進化が生物種の進化と違うのは、枝が再び交わることがあることだ。例えば Fortran の枝は Algol の子孫と融合しつつある。理論的には生物種でもあり得ることかもしれないが、その可能性は非常に小さく、今までに起こったことはたぶんなさそうだ。

Convergence is more likely for languages partly because the space of possibilities is smaller, and partly because mutations are not random. Language designers deliberately incorporate ideas from other languages.

言語で融合が起こりやすいのは、可能な空間が小さいからというのもあるし、変異がランダムでないということもあるだろう。言語の設計者は意図的に他の言語からアイデアを取り込む。

It's especially useful for language designers to think about where the evolution of programming languages is likely to lead, because they can steer accordingly. In that case, "stay on a main branch" becomes more than a way to choose a good language. It becomes a heuristic for making the right decisions about language design.

プログラミング言語の進化の向かう先について考えるのは、言語の設計者にとっては特に

有用だ。それによって方向を定めることができるからだ。この場合、「主要な枝に乗ってゆく」ことは良い言語を選ぶためだけでなく、言語設計において正しい選択を行うための良い経験則となるだろう。

Any programming language can be divided into two parts: some set of fundamental operators that play the role of axioms, and the rest of the language, which could in principle be written in terms of these fundamental operators.

どんなプログラミング言語も 2 つの部分に分けることができる。公理の役割を持つ根源的なオペレータと、原理的にそれらの根源的なオペレータを使って書くことができる残りの部分だ。

I think the fundamental operators are the most important factor in a language's long term survival. The rest you can change. It's like the rule that in buying a house you should consider location first of all. Everything else you can fix later, but you can't fix the location.

私が思うに、言語が長期間にわたって生き残るために最も重要な要素は、この根源的なオペレータだ。残りの部分は変えることができる。ちょうど、家を買うときには場所をまず熟考しなくちゃならないという法則と同じようなものだ。他の部分はいくらでも後から修正がきくが、場所だけは後から動かすわけにはいかない。

I think it's important not just that the axioms be well chosen, but that there be few of them. Mathematicians have always felt this way about axioms-- the fewer, the better-- and I think they're onto something.

さらに、公理は注意深く選ばれるだけでなく、なるべく少なくすることが重要だと思う。「少ないほうが良い」と、数学者は公理に関していつもそう感じてきた。そこには何か理由があると思う。

At the very least, it has to be a useful exercise to look closely at the core of a language to see if there are any axioms that could be weeded out. I've found in my long career as a slob that cruft breeds cruft, and I've seen this happen in software as well as under beds and in the corners of rooms.

少なくとも、取り除ける公理がないかどうか、言語のコアを注意深く調べることは有効な練習になるだろう。ものぐさ人間としての長いキャリアの間に私は、ホコリは次々とホコリを生んでいくということを発見したし、それがベッドの下や部屋の隅だけでなく、ソフ

トウェアにおいても起きるということを目にしてきた。

I have a hunch that the main branches of the evolutionary tree pass through the languages that have the smallest, cleanest cores. The more of a language you can write in itself, the better.

私の勘だが、進化樹の主要な枝は、最も小さくクリーンなコアを持つ言語を通っているのだと思う。言語の残りの部分をその言語自身で書くことができる余地が大きければ大きいほど、良い。

Of course, I'm making a big assumption in even asking what programming languages will be like in a hundred years. Will we even be writing programs in a hundred years? Won't we just tell computers what we want them to do?

もちろん、100 年後のプログラミング言語がどんなだろうと尋ねる時点で私はひとつの大きな仮定をおいている。100 年後には私たちはプログラムなんてしているのだろうか? してほしいことを言うだけでコンピュータがやってくれるようにならないのだろうか。

There hasn't been a lot of progress in that department so far. My guess is that a hundred years from now people will still tell computers what to do using programs we would recognize as such. There may be tasks that we solve now by writing programs and which in a hundred years you won't have to write programs to solve, but I think there will still be a good deal of programming of the type that we do today.

そっち方面では、これまでのところたいした進歩はない。私の推測では、100 年後でも、現在の私たちがプログラムと呼ぶようなものを使ってコンピュータにしてほしいことを伝えているだろう。もちろん、現在の私たちがプログラムを書いて解決している仕事のうち、プログラムを書かないでもよくなるものはあるだろうが、それでも現在の私たちがやっているようなプログラムはまだたくさん必要とされているだろうと思う。

It may seem presumptuous to think anyone can predict what any technology will look like in a hundred years. But remember that we already have almost fifty years of history behind us. Looking forward a hundred years is a graspable idea when we consider how slowly languages have evolved in the past fifty.

どんな技術であれ、100 年後を予想できるなんて考えるのは傲慢だと思われるかもしれない。しかし、私たちは既に 50 年の歴史を持っているということを考えてほしい。過去 50

年の言語の進化がいかによりゆっくりとしたものであるかを考えれば、100 年後を見るということも考え得る範囲だろう。

Languages evolve slowly because they're not really technologies. Languages are notation. A program is a formal description of the problem you want a computer to solve for you. So the rate of evolution in programming languages is more like the rate of evolution in mathematical notation than, say, transportation or communications. Mathematical notation does evolve, but not with the giant leaps you see in technology.

言語がゆっくりと進化するのは、それが本当は技術ではないからだ。言語は表記だ。プログラムは、コンピュータに解いてほしいと思う問題の正式な記述なんだ。だからプログラミング言語の進化の速度というのは、移動手段や通信手段よりは、数学表記の進化に近いだろう。数学の表記は確かに進化するが、技術の分野に見られるような巨大な飛躍はない。

Whatever computers are made of in a hundred years, it seems safe to predict they will be much faster than they are now. If Moore's Law continues to put out, they will be 74 quintillion (73,786,976,294,838,206,464) times faster. That's kind of hard to imagine. And indeed, the most likely prediction in the speed department may be that Moore's Law will stop working. Anything that is supposed to double every eighteen months seems likely to run up against some kind of fundamental limit eventually. But I have no trouble believing that computers will be very much faster. Even if they only end up being a paltry million times faster, that should change the ground rules for programming languages substantially. Among other things, there will be more room for what would now be considered slow languages, meaning languages that don't yield very efficient code.

100 年後にコンピュータが何で作られていようが、現在のものよりはるかに速い考えるのは安全な予測だろう。ムーアの法則がこのまま続けば、コンピュータは現在の 7400 京倍(73,786,976,294,838,206,464)の速度を得ることになる。想像し難い数字だ。実際、速度の面ではムーアの法則は破れるだろうと考えるほうが無難な予想だ。18 ヶ月ごとに 2 倍になるなんてものは、それが何であれ、いずれ根本的な限界に達しそうだと考えられる。そうだとすると、コンピュータが現在のものよりずっとずっと速くなると考えてよいだろう。たとえそれがたかだか 100 万倍であったとしても、プログラミング言語の基本的な法則を大きく変えるには十分だ。その中には、例えば現在では遅いと考えられている言語、つまり効率的なコードを生成しない言語の活躍の余地が大きくなるということが挙げられ

る。

And yet some applications will still demand speed. Some of the problems we want to solve with computers are created by computers; for example, the rate at which you have to process video images depends on the rate at which another computer can generate them. And there is another class of problems which inherently have an unlimited capacity to soak up cycles: image rendering, cryptography, simulations.

もちろん、依然として速度を要求するアプリケーションもあるだろう。コンピュータを使って解きたいと思う問題のいくつかは、コンピュータ自身によって作られる。例えば、コンピュータで生成されたビデオイメージの処理は、生成側のコンピュータの速度に追いついていなければならないだろう。さらに、いくらマシンサイクルがあっても本質的にそれを喰い潰してしまう類の処理というのがある。イメージレンダリング、暗号、シミュレーションといったものだ。

If some applications can be increasingly inefficient while others continue to demand all the speed the hardware can deliver, faster computers will mean that languages have to cover an ever wider range of efficiencies. We've seen this happening already. Current implementations of some popular new languages are shockingly wasteful by the standards of previous decades.

いくつかのアプリケーションはますます非効率になって行く一方で、ハードウェアの限界までの速度を要求するアプリケーションがあるとすると、速い計算機を手にするということは、言語は効率においてより広い範囲をカバーしなければならなくなるということだ。既にそれは起こりつつある。人気のある新しい言語の現在の実装は、10年前の基準では驚異的に非効率だ。

This isn't just something that happens with programming languages. It's a general historical trend. As technologies improve, each generation can do things that the previous generation would have considered wasteful. People thirty years ago would be astonished at how casually we make long distance phone calls. People a hundred years ago would be even more astonished that a package would one day travel from Boston to New York via Memphis.

これはプログラミング言語だけに起こっていることではない。一般的な歴史の流れだ。技術が進むにつれ、各世代はその前の世代だっただけでもったいないと思っていたようなことを平気でできるようになる。30年前の人は、私たちがこんなに気軽に長距離電話をかけるのに驚愕するだろう。100年前の人は、ボストンからニューヨークまで翌日配達便で荷物を

送るのに、メンフィスを経由すると聞いてひっくり返るだろう。

I can already tell you what's going to happen to all those extra cycles that faster hardware is going to give us in the next hundred years. They're nearly all going to be wasted.

次の 100 年の間に、速いハードウェアによって私たちが得る余分なマシンサイクルがどう使われるかを当ててみようか。そのほとんどは無駄に使われることになる。

I learned to program when computer power was scarce. I can remember taking all the spaces out of my Basic programs so they would fit into the memory of a 4K TRS-80. The thought of all this stupendously inefficient software burning up cycles doing the same thing over and over seems kind of gross to me. But I think my intuitions here are wrong. I'm like someone who grew up poor, and can't bear to spend money even for something important, like going to the doctor.

私は、コンピュータの力が貴重であった時代にプログラミングを学んだ。4K バイトメモリ の TRS-80 に載せるために、Basic プログラムから全ての空白を抜いたことを覚えている。同じ作業を何度も何度も繰り返すのにマシンサイクルを喰い続ける、おそろしく非効率なソフトウェアのことを考えると気分が悪くなる。だが、ここでは私の直観は間違っているんだろう。貧乏が身に付いてしまうと、例えば医者にかかるといった重要なことに金を使うのを躊躇してしまうようなものだ。

Some kinds of waste really are disgusting. SUVs, for example, would arguably be gross even if they ran on a fuel which would never run out and generated no pollution. SUVs are gross because they're the solution to a gross problem. (How to make minivans look more masculine.) But not all waste is bad. Now that we have the infrastructure to support it, counting the minutes of your long-distance calls starts to seem niggling. If you have the resources, it's more elegant to think of all phone calls as one kind of thing, no matter where the other person is.

本当に気分が悪くなるような無駄があることは確かだ。SUV は、たとえそれが無公害で永遠になくならないような燃料で走ったとしたって、気持ちの悪いものじゃない。それというのも、SUV 自身が醜い問題への回答だからだ(ミニバンを何とかして逞しく見せようっていうね)。でも、全ての無駄が悪いわけじゃない。インフラが整った現在では、長距離電話をかける最中に時間を分刻みで気にするなんて些細なことのようになってくる。資源があれば、相手がどこにしようと電話をかけるというのはひとつの均一な行為だと考えるほう

がエレガントだ。

There's good waste, and bad waste. I'm interested in good waste-- the kind where, by spending more, we can get simpler designs. How will we take advantage of the opportunities to waste cycles that we'll get from new, faster hardware?

良い無駄と悪い無駄があるってことだ。私は良い無駄の方に関心がある。贅沢に使うことで、より単純なデザインが得られるような無駄だ。新しい、速いハードウェアのマシンサイクルを無駄に使えるというチャンスをどんなふうに利用できるだろう。

The desire for speed is so deeply engrained in us, with our puny computers, that it will take a conscious effort to overcome it. In language design, we should be consciously seeking out situations where we can trade efficiency for even the smallest increase in convenience.

貧弱なコンピュータを使っているうちに、速度への渴望は私たちの中にあまりに深く刻み込まれてしまったため、それを乗り越えるには意識的な努力が必要だ。言語の設計においても、少しでも便利にするために効率を犠牲にできるところがないかどうか、意識的に探さなければならない。

Most data structures exist because of speed. For example, many languages today have both strings and lists. Semantically, strings are more or less a subset of lists in which the elements are characters. So why do you need a separate data type? You don't, really. Strings only exist for efficiency. But it's lame to clutter up the semantics of the language with hacks to make programs run faster. Having strings in a language seems to be a case of premature optimization.

多くのデータ構造は速度のために存在する。例えば、現代の言語の多くは文字列とリストを持っている。意味的には、文字列は、全ての要素が文字であるというリストであって、一般的なリストのサブセットと言える。ならどうして別のデータタイプが必要なんだろう。実は必要じゃないんだ。文字列は単に効率のためだけに存在している。けれど、プログラムを速く走らせるハックのためだけに言語に意味をゴテゴテと付け足すなんて、スマートじゃない。言語の中に文字列が存在すること自体が早過ぎる最適化の事例のようだ。

If we think of the core of a language as a set of axioms, surely it's gross to have additional axioms that add no expressive power, simply for the sake of efficiency. Efficiency is important, but I don't think that's the right way to get it.

言語のコアを公理の集合と考えると、何ら表現能力を増やさない公理を効率のためだけに追加するというのは醜いことだ。効率は重要だが、公理を追加することで達成するのは正しいやり方とは思えない。

The right way to solve that problem, I think, is to separate the meaning of a program from the implementation details. Instead of having both lists and strings, have just lists, with some way to give the compiler optimization advice that will allow it to lay out strings as contiguous bytes if necessary.

正しいやり方は、プログラムの意味と実装の詳細を分けることだ。リストと文字列を持つ代わりに、リストだけにしておいて、コンパイラが必要なら文字列を連続したバイトとして表現できるような最適化のアドバイスを与えられるようにしておくんだ。

Since speed doesn't matter in most of a program, you won't ordinarily need to bother with this sort of micromanagement. This will be more and more true as computers get faster.

たいていの、速度が問題じゃないプログラムにおいては、こんな細かいことを気にする必要はないだろう。コンピュータが速くなればなるほど、そういう場合は増えてくる。

Saying less about implementation should also make programs more flexible. Specifications change while a program is being written, and this is not only inevitable, but desirable.

実装を気にして書く必要が少なくなれば、プログラムはより柔軟になる。プログラムを書いている最中に仕様が変わっても対応できる。これは単に避け難いというだけでなく、望ましいことだ。

The word "essay" comes from the French verb "essayer", which means "to try". An essay, in the original sense, is something you write to try to figure something out. This happens in software too. I think some of the best programs were essays, in the sense that the authors didn't know when they started exactly what they were trying to write.

「エッセイ」という単語はフランス語の動詞"essayer"から来ている。それは「試す」という意味だ。エッセイは、もともとの意味においては、何かを理解するために書いてみるものだった。ソフトウェアも同じことだ。最良のプログラムのいくつかは、作者が何を書くべきかはっきり分からないままに書き始めたという意味で、エッセイであると言えるだろう。

う。

Lisp hackers already know about the value of being flexible with data structures. We tend to write the first version of a program so that it does everything with lists. These initial versions can be so shockingly inefficient that it takes a conscious effort not to think about what they're doing, just as, for me at least, eating a steak requires a conscious effort not to think where it came from.

Lisp ハッカーは既にデータ構造を柔軟に持つことの価値を知っている。最初のバージョンでは何もかもリストで処理するように書くことが多い。こういった最初のバージョンはあまりにもおそろしく非効率なんで、そのプログラムが走っている時に何をしているのかを考えないようにするには意識的な努力が必要だ。私は、ステーキを食べている時にそれがどこから来たかをなるべく考えないようにしているが、それと同じようなものだ。

What programmers in a hundred years will be looking for, most of all, is a language where you can throw together an unbelievably inefficient version 1 of a program with the least possible effort. At least, that's how we'd describe it in present-day terms. What they'll say is that they want a language that's easy to program in.

100 年後のプログラマが求めているものは、たいていの場合、信じられないほど非効率な最初のバージョンのプログラムを、最低限の努力ででっちあげられるような言語だろう。現在の我々の言葉を使えば、だが。彼らに言わせれば、単に「簡単にプログラムできる言語」ということになるだろう。

Inefficient software isn't gross. What's gross is a language that makes programmers do needless work. Wasting programmer time is the true inefficiency, not wasting machine time. This will become ever more clear as computers get faster.

非効率なソフトウェアが醜いのではない。醜いのは、プログラマに不要な仕事をさせる言語だ。本当の非効率性とは、マシンの時間を無駄にすることではなく、プログラマの時間を無駄にすることだ。コンピュータが速くなればなるほど、このことははっきりしてくる。

I think getting rid of strings is already something we could bear to think about. We did it in [Arc](#), and it seems to be a win; some operations that would be awkward to describe as regular expressions can be described easily as recursive functions.

文字列を取り除くことは、現在の私たちにも何とか考え得る範疇にあると思う。私たちは既に **Arc** でそれをやってみて、結果は満足 of いくものとなっている。正規表現のような不器用な記述を要する処理も、再帰関数で簡単に書ける。

How far will this flattening of data structures go? I can think of possibilities that shock even me, with my conscientiously broadened mind. Will we get rid of arrays, for example? After all, they're just a subset of hash tables where the keys are vectors of integers. Will we replace hash tables themselves with lists?

どこまでデータ構造をこんな具合に畳んでゆけるだろう?意識的に想像力を押し広げることで、自分でも驚くくらいの可能性を考えてみることができる。例えば、配列を取り除くことはできるだろうか。結局のところ、配列はハッシュテーブルのキーが整数のベクターであるものにすぎない。さらに、ハッシュテーブルそのものをリストで置き換えることができるだろうか。

There are more shocking prospects even than that. The Lisp that McCarthy described in 1960, for example, didn't have numbers. Logically, you don't need to have a separate notion of numbers, because you can represent them as lists: the integer n could be represented as a list of n elements. You can do math this way. It's just unbearably inefficient.

もっと驚くべき予測もできる。マッカーシーが 1960 年に記述した **Lisp** には数値がなかった。理論的には、数値というデータ型を別に持つ必要はない。リストで表現できるからだ。整数 n は n 個の要素を持つリストで表現できる。それを使って計算もできる。ただ、耐え難いほど非効率なだけだ。

No one actually proposed implementing numbers as lists in practice. In fact, McCarthy's 1960 paper was not, at the time, intended to be implemented at all. It was a [theoretical exercise](#), an attempt to create a more elegant alternative to the Turing Machine. When someone did, unexpectedly, take this paper and translate it into a working Lisp interpreter, numbers certainly weren't represented as lists; they were represented in binary, as in every other language.

数値をリストで表現しようなんて実際に提案した人はいなかった。事実、マッカーシーの 1960 年の論文は、当時は実装を意図したものではなかったんだ。それはチューリングマシンのよりスマートな代替を作ろうという理論的な演習であった。その論文が予想外にも実際に動作する **Lisp** インタプリタとして実装された時、数値はリストで表されたわけではなかった。他の言語と同様に、2 進数で表されていたんだ。

Could a programming language go so far as to get rid of numbers as a fundamental data type? I ask this not so much as a serious question as as a way to play chicken with the future. It's like the hypothetical case of an irresistible force meeting an immovable object-- here, an unimaginably inefficient implementation meeting unimaginably great resources. I don't see why not. The future is pretty long. If there's something we can do to decrease the number of axioms in the core language, that would seem to be the side to bet on as t approaches infinity. If the idea still seems unbearable in a hundred years, maybe it won't in a thousand.

プログラミング言語は基礎的なデータ型から数値を取り除くというところまで行けるだろうか。これは真剣な質問というより、むしろ未来にどこまで行けるかを試してみる実験だ。動かせない物体に抗えない力を加えたらどうなるかという思考実験のようなものだ。信じ難いほど非効率な実装が、信じ難いほど膨大な資源を使えたとしたら…ありえなくはないんじゃないか。先は長いんだ。言語のコアの公理の数を減らすことができるものがあるとしたら、 t が無限大に近づくにつれ、いずれはそちらの方に向かってゆくんじゃないだろうか。100 年後にまだこのアイディアが許容できないとしたって、1000 年後はどうなるか分からないよ。

Just to be clear about this, I'm not proposing that all numerical calculations would actually be carried out using lists. I'm proposing that the core language, prior to any additional notations about implementation, be defined this way. In practice any program that wanted to do any amount of math would probably represent numbers in binary, but this would be an optimization, not part of the core language semantics.

一応断っておくが、私は何も数値計算を全てリストでやろうと提案しているわけじゃない。実装に必要な追加の表記を加える前の、コアの言語はこういうふうに定義すべきじゃないかと提案しているんだ。現実には、数値計算を行うプログラムは数値を 2 進数で表現するだろうけど、それは最適化と見なされるべきで、言語のコアの意味として考えられるべきじゃない。

Another way to burn up cycles is to have many layers of software between the application and the hardware. This too is a trend we see happening already: many recent languages are compiled into byte code. Bill Woods once told me that, as a rule of thumb, each layer of interpretation costs a factor of 10 in speed. This extra

cost buys you flexibility.

マシンサイクルを浪費するもうひとつの方法は、ハードウェアとアプリケーションの間にたくさんの層を作ることだ。この流行は既に始まっている。多くの新しい言語はバイトコードへとコンパイルされる。ビル・ウッズは、大まかな目安として、1つ層を追加するとだいたい10倍遅くなると私に言ったことがある。この追加のコストを払って柔軟性を得ているわけだ。

The very first version of Arc was an extreme case of this sort of multi-level slowness, with corresponding benefits. It was a classic "metacircular" interpreter written on top of Common Lisp, with a definite family resemblance to the eval function defined in McCarthy's original Lisp paper. The whole thing was only a couple hundred lines of code, so it was very easy to understand and change. The Common Lisp we used, CLisp, itself runs on top of a byte code interpreter. So here we had two levels of interpretation, one of them (the top one) shockingly inefficient, and the language was usable. Barely usable, I admit, but usable.

最初のバージョンの Arc はこの多層による遅いプログラムの極端な例で、それに付随する利点もあった。それは古典的な「超循環」インタプリタで、Common Lisp で書かれており、マッカーシーのものの Lisp 論文の eval 関数に極めて似たものだった。全てはほんの200行ほどのコードで、だからすぐ理解できたし変更も容易だった。私たちが使った Common Lisp は CLisp で、それ自身がバイトコードインタプリタで動作していた。したがって私たちは2つのインタプリタの層を持っていて、しかもそのひとつ(上の層)はおそろしく非効率だったのだが、それでも言語を使うことはできた。かろうじて、ではあったが。

Writing software as multiple layers is a powerful technique even within applications. Bottom-up programming means writing a program as a series of layers, each of which serves as a language for the one above. This approach tends to yield smaller, more flexible programs. It's also the best route to that holy grail, reusability. A language is by definition reusable. The more of your application you can push down into a language for writing that type of application, the more of your software will be reusable.

ソフトウェアを複数の層に分けて書くのはアプリケーション内でも強力なテクニックだ。ひとつの層を次の層の記述言語として使って、プログラムを層の積み重ねで構成してゆくの、ボトムアッププログラミングという。このアプローチを取ると、多くの場合プログラムはより小さく、柔軟となる。またこの手法は、再利用性という聖杯に至る最良の路でもある。言語は、その定義からして、再利用されるものだ。アプリケーションのなるべく

多くの部分を、そういう種類のアプリケーションを書くための言語に落とし込んでゆくことで、あなたのソフトウェアはより再利用がしやすくなる。

Somehow the idea of reusability got attached to object-oriented programming in the 1980s, and no amount of evidence to the contrary seems to be able to shake it free. But although some object-oriented software is reusable, what makes it reusable is its bottom-upness, not its object-orientedness. Consider libraries: they're reusable because they're language, whether they're written in an object-oriented style or not.

どういうわけか、1980年代に、再利用性という概念はオブジェクト指向プログラミングとくっついてしまって、どんなに反証を挙げてもそこからひっぺがすことができなくなってしまったようだ。確かにオブジェクト指向ソフトウェアのなかには再利用性の高いものがあるが、それはオブジェクト指向だからそうなのではなく、ボトムアップ性があるからなんだ。ライブラリを考えてみたまえ。オブジェクト指向スタイルで書かれていようがいまいが、それが言語であるからこそ、再利用できるんじゃないか。

I don't predict the demise of object-oriented programming, by the way. Though I don't think it has much to offer good programmers, except in certain specialized domains, it is irresistible to large organizations. Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches. Large organizations always tend to develop software this way, and I expect this to be as true in a hundred years as it is today.

私はオブジェクト指向プログラミングの終焉を予言しようとしているんじゃないよ。オブジェクト指向が良いプログラマに与えてくれるものは、いくつかの特殊な領域を除いてはたいしてないと思っているけれど、大きな組織にとっては抗い難い魅力があるのだろう。オブジェクト指向プログラミングはスパゲッティコードを書くための持続的な方法を提供してくれる。プログラムをパッチの繰り返しで書くことを加速するんだ。大きな組織はソフトウェアをいつでもそういうふうに書いてきたし、100年経ってもそれは変わらないだろう。

As long as we're talking about the future, we had better talk about parallel computation, because that's where this idea seems to live. That is, no matter when you're talking, parallel computation seems to be something that is going to happen in the future.

将来の話をするなら、並列計算にも触れておくべきだろう。将来こそが並列計算にふさわしい場所だからだ。つまり、いつの時代であっても、並列計算とは将来に来るものでありつづけるだろうということなんだが。

Will the future ever catch up with it? People have been talking about parallel computation as something imminent for at least 20 years, and it hasn't affected programming practice much so far. Or hasn't it? Already chip designers have to think about it, and so must people trying to write systems software on multi-cpu computers.

未来はいつか並列計算に追い付くんだろうか?もうすぐ並列計算が来る、と人々は少なくともこの 20 年は言い続けてきた。それでいてプログラミング作法にはあまり影響を与えているようには思えない。それとも影響はあるのか?確かにチップ設計者は並列計算を考慮しなければならなくなっているし、マルチ CPU のコンピュータのシステムソフトウェアを書こうとする人もそうだ。

The real question is, how far up the ladder of abstraction will parallelism go? In a hundred years will it affect even application programmers? Or will it be something that compiler writers think about, but which is usually invisible in the source code of applications?

本当に知りたいのは、並列計算が抽象化の梯子をどのくらいまで登ってゆくだろうかということだ。100 年後にはアプリケーションプログラマでさえも影響を受けるだろうか。それともコンパイラを書く人だけが気にしていて、アプリケーションのソースコードからは見えないものになるだろうか。

One thing that does seem likely is that most opportunities for parallelism will be wasted. This is a special case of my more general prediction that most of the extra computer power we're given will go to waste. I expect that, as with the stupendous speed of the underlying hardware, parallelism will be something that is available if you ask for it explicitly, but ordinarily not used. This implies that the kind of parallelism we have in a hundred years will not, except in special applications, be massive parallelism. I expect for ordinary programmers it will be more like being able to fork off processes that all end up running in parallel.

ひとつ起こりそうなことは、並列化の機会の多くは無駄にされるだろうということだ。これは、余分な計算能力の大部分が無駄になるだろうというの一般的な予測の特別な場合にすぎない。基礎となるハードウェアがものすごく速くなった時代には、並列性ははっきり要求すれば使えるが、そうでなければ使われないというものになるだろうと思っている。

したがって、特別なアプリケーションを除いては、100 年後に使われている並列計算は超並列計算ではないだろう。普通のプログラマにとっては、プロセスを気軽に作ってそれがたまたま並列に走るとか、そういったものになるだろう。

And this will, like asking for specific implementations of data structures, be something that you do fairly late in the life of a program, when you try to optimize it. Version 1s will ordinarily ignore any advantages to be got from parallel computation, just as they will ignore advantages to be got from specific representations of data.

そして並列計算も、特定のデータ構造の実装を指定するのと同じく、プログラム開発のずっと後半、最適化を行う段階で導入するものになるだろう。バージョン 1 は、特定のデータ表現から得られる利点を無視するのと同じように、並列計算の利点を全く無視したものになるだろう。

Except in special kinds of applications, parallelism won't pervade the programs that are written in a hundred years. It would be premature optimization if it did. 特別な種類のアプリケーションを除いて、並列計算は 100 年後のプログラムにさほど影響を与えないだろう。もし与えていたら、それは早過ぎる最適化になるだろう。

How many programming languages will there be in a hundred years? There seem to be a huge number of new programming languages lately. Part of the reason is that faster hardware has allowed programmers to make different tradeoffs between speed and convenience, depending on the application. If this is a real trend, the hardware we'll have in a hundred years should only increase it.

100 年後には、プログラミング言語はいくつくらい存在するだろうか。最近になってすごい数のプログラミング言語が生まれてきた。理由のひとつは、ハードウェアが速くなったことで、プログラマが速度と利便性の間にアプリケーションに応じて異なったトレードオフを求められるようになったということだ。この流れが本物ならば、100 年後のハードウェアはこの傾向を後押しするだけだろう。

And yet there may be only a few widely-used languages in a hundred years. Part of the reason I say this is optimism: it seems that, if you did a really good job, you could make a language that was ideal for writing a slow version 1, and yet with the right optimization advice to the compiler, would also yield very fast code when

necessary. So, since I'm optimistic, I'm going to predict that despite the huge gap they'll have between acceptable and maximal efficiency, programmers in a hundred years will have languages that can span most of it.

だが、100年後でも、広く使われる言語は2〜3個ではないかと私には思われる。私の楽観性もあるのだが、もしうまくやれば、遅いバージョン 1 を書くのにも、また適切なアドバンスをコンパイラに与えることによって必要なら非常に速いコードを生成するのにも、理想的な言語というのを作れるんじゃないかと思っているからだ。私は楽観的だから、まああまの効率と最高の効率との乖離が非常に大きくなるであろう時代でも、その多くの部分をカバーできる言語ができるだろうと予測する。

As this gap widens, profilers will become increasingly important. Little attention is paid to profiling now. Many people still seem to believe that the way to get fast applications is to write compilers that generate fast code. As the gap between acceptable and maximal performance widens, it will become increasingly clear that the way to get fast applications is to have a good guide from one to the other. この乖離が大きくなるにつれ、プロファイラの重要性は増してゆく。現在、プロファイリングに関してはほとんど注意が払われていない。たくさんの人々が、速いアプリケーションを得る方法は速いコードを出すコンパイラを書くことだと、依然として信じている。まああまの効率と最高の効率との乖離が進むにつれ、前者から後者へと至る良いガイドを持つことが速いアプリケーションを得る方法だということがはっきりしてくるだろう。

When I say there may only be a few languages, I'm not including domain-specific "little languages". I think such embedded languages are a great idea, and I expect them to proliferate. But I expect them to be written as thin enough skins that users can see the general-purpose language underneath.

私は言語が2〜3個になるだろうと言ったが、特定の領域に特化された「小さな言語」は含めていない。そういう埋め込み言語は良いアイディアだと思うし、きっと多くが生まれてくることだろう。だが、それらの言語は、汎用の実装言語が透けてみえるくらいのごく薄い表皮として書かれるようになるんじゃないかと思う。

Who will design the languages of the future? One of the most exciting trends in the last ten years has been the rise of open-source languages like Perl, Python, and Ruby. Language design is being taken over by hackers. The results so far are messy, but encouraging. There are some stunningly novel ideas in Perl, for

example. Many are stunningly bad, but that's always true of ambitious efforts. At its current rate of mutation, God knows what Perl might evolve into in a hundred years.

未来の言語は誰が設計するだろう。過去 10 年間の流れのうちもっともエキサイティングなもののひとつは、Perl、Python、Ruby といったオープンソース言語の隆盛だ。言語の設計がハッカーの手に渡ったんだ。今のところ、得られている結果はまだごちゃごちゃとしているが、期待できる。例えば Perl にはいくつか、驚くほど新しいアイデアが見て取れる。そのうちの大部分は驚くほど悪いが、それは野心的な努力にはつきものだ。そして現在の変異の速度を考えるに、100 年後に Perl がどうなっているかなんて誰にも分からない。

It's not true that those who can't do, teach (some of the best hackers I know are professors), but it is true that there are a lot of things that those who teach can't do. [Research](#) imposes constraining caste restrictions. In any academic field there are topics that are ok to work on and others that aren't. Unfortunately the distinction between acceptable and forbidden topics is usually based on how intellectual the work sounds when described in research papers, rather than how important it is for getting good results. The extreme case is probably literature; people studying literature rarely say anything that would be of the slightest use to those producing it.

実行することができない者が教えるのだ、というのは正しくない(私の知る最高のハッカーの幾人かは大学教授だ)。しかし、教えることのできる者が実行できないことがあるというのは正しいだろう。リサーチは厳しいカーストの制限を受ける。どんな学問領域でも、研究してよいテーマとそうでないテーマがある。残念ながら、研究してよいテーマと禁じられたテーマの差異は、良い結果を得るためにどれだけ重要かではなく、論文にした時にどれだけ知的に見えるかということで測られることが多い。最も極端なのはおそらく文学だろう。文学を研究している人々が、実際に文学を創り出している人々にちょっとでも役に立つことを言うことはほとんどない。

Though the situation is better in the sciences, the overlap between the kind of work you're allowed to do and the kind of work that yields good languages is distressingly small. (Olin Shivers has grumbled eloquently about this.) For example, types seem to be an inexhaustible source of research papers, despite the fact that static typing seems to preclude true macros-- without which, in my opinion, no language is worth using.

科学ではまだ状況はましだが、研究でやれることと良い言語を作ることの共通部分はがっかりするほど小さい。(オリン・シヴァーズがかつてこのことに対する不満を雄弁に語って

いた)。例えば、型は尽きせぬ論文の宝庫のようだが、静的型付けは本物のマクロを排除するように見える。そして私の意見では、本物のマクロのない言語なんて使うに値しない。

The trend is not merely toward languages being developed as open-source projects rather than "research", but toward languages being designed by the application programmers who need to use them, rather than by compiler writers. This seems a good trend and I expect it to continue.

言語が「研究」ではなくオープンソースプロジェクトとして開発されつつあるという流れの他に、コンパイラ開発者でなく実際に言語を使うアプリケーションプログラマが言語を設計するようになってきたという流れがある。これは良いことだと思うし、続いていってほしい。

Unlike physics in a hundred years, which is almost necessarily impossible to predict, I think it may be possible in principle to design a language now that would appeal to users in a hundred years.

100 年後の物理学は必然的にほとんど予測不可能だが、100 年後のユーザを惹きつける言語を現在設計することは、原理的に可能だと私は考える。

One way to design a language is to just write down the program you'd like to be able to write, regardless of whether there is a compiler that can translate it or hardware that can run it. When you do this you can assume unlimited resources. It seems like we ought to be able to imagine unlimited resources as well today as in a hundred years.

言語を設計するひとつの方法は、その言語のコンパイラやそれを走らせるハードウェアがあるかないかということを考えずに、こういうプログラムが書きたいんだ、というプログラムを書いてみることだ。無制限の資源があると思って書いてみよう。100 年後でなく現在でも、無制限の資源を想像することはできるはずだ。

What program would one like to write? Whatever is least work. Except not quite: whatever *would be* least work if your ideas about programming weren't already influenced by the languages you're currently used to. Such influence can be so pervasive that it takes a great effort to overcome it. You'd think it would be obvious to creatures as lazy as us how to express a program with the least effort. In fact, our ideas about what's possible tend to be so [limited](#) by whatever

language we think in that easier formulations of programs seem very surprising. They're something you have to discover, not something you naturally sink into.

どんなプログラムを書きたいだろう。なるべく少ない手間で書けるものだろう。ただし、あなたが現在使い慣れた言語の影響をなくした上で、少ない手間ということを考える必要がある。そのような影響はあまりに深くて、乗り越えるには大きな努力が必要だ。私たちのような怠惰な生物にとって、プログラムを最低限の努力で表現する方法は明白だと思うかもしれない。実際は、何が可能かという点についての私たちの考えは、私たちが考える言語によってあまりに制限されている。だから、プログラムの簡単な記述法とは、当初予想もしなかったものになるだろう。自然にそこに到るようなものではなく、発見しなければならないのだ。

One helpful trick here is to use the [length](#) of the program as an approximation for how much work it is to write. Not the length in characters, of course, but the length in distinct syntactic elements-- basically, the size of the parse tree. It may not be quite true that the shortest program is the least work to write, but it's close enough that you're better off aiming for the solid target of brevity than the fuzzy, nearby one of least work. Then the algorithm for language design becomes: look at a program and ask, is there any way to write this that's shorter?

有効な手法のひとつは、プログラムの長さを、それを書くのに必要な仕事量の目安に使うことだ。ここでいう長さとは、もちろん文字数じゃなくて、構文要素の長さだ。基本的には構文木の大きさということになる。最も短いプログラムを書くのが最も楽だというのは必ずしも正しくないだろうが、なんとなく楽な方法というのを探し求めるよりは、明確な目標となり得るだろう。すると、言語設計のアルゴリズムはこんなふうになる。つまり、プログラムを見て、これをもっと短く書く方法はないかと考えるのだ。

In practice, writing programs in an imaginary hundred-year language will work to varying degrees depending on how close you are to the core. Sort routines you can write now. But it would be hard to predict now what kinds of libraries might be needed in a hundred years. Presumably many libraries will be for domains that don't even exist yet. If SETI@home works, for example, we'll need libraries for communicating with aliens. Unless of course they are sufficiently advanced that they already communicate in XML.

実際には、この仮想的な 100 年後の言語でプログラムを書くという行為は、どのくらい言語のコアに近いものを書いているかで大きく異なるだろう。ソートルーチンなら今でも書ける。でも、100 年後にどんなライブラリが必要になるかを予測するのは難しい。たくさんのライブラリは現在には存在さえもしていない領域のものになるだろうから。例えば、

もし SETI@home が成功したならば、異星人と通信するためのライブラリが必要になるかもしれない。もちろん彼ら異星人が十分に進化していて、XML で通信し合っていなければ、の話だが。

At the other extreme, I think you might be able to design the core language today. In fact, some might argue that it was already mostly designed in 1958.

逆の方向として、コア言語だけなら現在でも設計できるかもしれない。実際、それは 1958 年に既になされたと主張する人さえいるだろう。

If the hundred year language were available today, would we want to program in it? One way to answer this question is to look back. If present-day programming languages had been available in 1960, would anyone have wanted to use them? 100 年後の言語が今存在するとして、それでプログラムしたいと思うだろうか。この問いに答える方法のひとつは、過去を振り返ってみることだ。もし現在のプログラミング言語が 1960 年に存在したとして、それでプログラムしたいと思う人がいただろうか。

In some ways, the answer is no. Languages today assume infrastructure that didn't exist in 1960. For example, a language in which indentation is significant, like Python, would not work very well on printer terminals. But putting such problems aside-- assuming, for example, that programs were all just written on paper-- would programmers of the 1960s have liked writing programs in the languages we use now?

ある意味で答えはノーだ。今日の言語は 1960 年には存在しなかったインフラを仮定している。例えばインデントに意味がある Python のような言語は、プリンタ端末では使いづらいだろう。しかしそういう問題を脇において、例えば、プログラムは全部紙に書かれるものだと、1960 年代のプログラマは現在私たちが使っているプログラムで書くことを好むだろうか？

I think so. Some of the less imaginative ones, who had artifacts of early languages built into their ideas of what a program was, might have had trouble. (How can you manipulate data without doing pointer arithmetic? How can you implement flow charts without gotos?) But I think the smartest programmers would have had no trouble making the most of present-day languages, if they'd had them.

私はそう思う。想像力が十分でなく、初期の言語の制限によってプログラムに関する考え

が縛られてしまっている人には難しいかもしれない。(ポインタ演算なしでどうやってデータを操作するんだ?goto なしでどうやってフローチャートを実装するんだ?)しかし、賢いプログラマなら、与えられれば現在のプログラミング言語を使いこなすことに何ら困難を感じないだろうと思う。

If we had the hundred-year language now, it would at least make a great pseudocode. What about using it to write software? Since the hundred-year language will need to generate fast code for some applications, presumably it could generate code efficient enough to run acceptably well on our hardware. We might have to give more optimization advice than users in a hundred years, but it still might be a net win.

100 年後の言語が今あったとすれば、少なくとも疑似コードを書くのに素晴らしい手段となるだろう。それを使ってソフトウェアを書いてみるのはどうか。100 年後の言語は少なくともある種のアプリケーションでは速いコードを生成するはずだから、うまくすると現在のハードウェアでもどうにか走るくらいのコードが生成できるかもしれない。100 年後よりはたくさんの最適化のアドバイスを与えなければならないかもしれないが、総合的に見ればそのほうが良いかもしれない。

Now we have two ideas that, if you combine them, suggest interesting possibilities: (1) the hundred-year language could, in principle, be designed today, and (2) such a language, if it existed, might be good to program in today. When you see these ideas laid out like that, it's hard not to think, why not try writing the hundred-year language now?

さあ、ここに 2 つのアイデアが出た。これを組み合わせると面白い可能性が見えてくる。(1)100 年後の言語は、原理的には、現在でも設計できる。(2)そのような言語は、もし存在すれば、現在でもプログラムを書くのに良い言語かもしれない。こんなふうにアイデアを並べて見ていると、100 年後の言語を、今、設計したくなってこないかい?

When you're working on language design, I think it is good to have such a target and to keep it consciously in mind. When you learn to drive, one of the principles they teach you is to align the car not by lining up the hood with the stripes painted on the road, but by aiming at some point in the distance. Even if all you care about is what happens in the next ten feet, this is the right answer. I think we can and should do the same thing with programming languages.

言語を設計する時には、そういうターゲットを設定して、常に頭に置いておくことは良いことだと思う。運転を習う時、道路の線に車を合わせるのではなく、遠くの点を目標にして方向を合わせるとよいと教わるはずだ。たとえ目先 **10** フィートまでに起こることが重要なのだとしても、そのアドバイスは正しい。プログラミング言語に関しても、それは同じだ。

Notes

注釈

I believe Lisp Machine Lisp was the first language to embody the principle that declarations (except those of dynamic variables) were merely optimization advice, and would not change the meaning of a correct program. Common Lisp seems to have been the first to state this explicitly.

Lisp Machine Lisp が、(動的変数以外の)宣言は最適化のためのアドバイスであって正しいプログラムの意味を変えるものではない、という原理を採用した、最初の言語であったと思う。Common Lisp はそれを明示した最初のものだろう。

Thanks to Trevor Blackwell, Robert Morris, and Dan Giffin for reading drafts of this, and to Guido van Rossum, Jeremy Hylton, and the rest of the Python crew for inviting me to speak at PyCon.

Chapter 12 BEATING THE AVERAGES

第 12 章 普通のやつらの上を行け

April 2001, rev. April 2003

(This article is derived from a talk given at the 2001 Franz Developer Symposium.)

(この記事は 2001 年に開催された Franz Developer Symposium で行った講演をベースにしている。)

In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface.

1995 年の夏、私は友人のロバート・モリスとともに Viaweb というベンチャー企業を立ち上げた。エンドユーザがオンラインストアを自分で作ることが出来るソフトウェアを書く、というのが計画だった。当時、このソフトウェアが新しかったのは、ソフトウェア自身を自分たちのサーバで走らせて、通常の Web ページをインタフェースにするという点だった。

A lot of people could have been having this idea at the same time, of course, but as far as I know, Viaweb was the first Web-based application. It seemed such a novel idea to us that we named the company after it: Viaweb, because our software worked via the Web, instead of running on your desktop computer.

もちろんたくさんの人たちがこのアイディアを思い付いていたのだろうが、私の知る限り、Viaweb は最初の Web ベースアプリケーションだった。そのアイディアがとても斬新なものに思えたので、私たちは自分の会社をそのように名付けたくらいだ。私たちのソフトウェアはユーザのデスクトップではなく、Web を通して(via web)動く、というわけだ。

Another unusual thing about this software was that it was written primarily in a programming language called Lisp. It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs. [\[1\]](#)

もうひとつ、私たちのソフトウェアがユニークだったのは、それが主に Lisp と呼ばれるプログラミング言語で書かれていたからだ。それまで主として大学や研究所で使われていた Lisp の、事実上初めての大規模なエンドユーザアプリケーションだった。 [\[1\]](#)

The Secret Weapon

秘密兵器

Eric Raymond has written an essay called "How to Become a Hacker," and in it, among other things, he tells would-be hackers what languages they should learn. He suggests starting with Python and Java, because they are easy to learn. The serious hacker will also want to learn C, in order to hack Unix, and Perl for system administration and cgi scripts. Finally, the truly serious hacker should consider learning Lisp:

エリック・レイモンドはエッセイ「ハッカーになろう」の中で、他のいろいろなアドバイスに混じって、ハッカーになりたい人はどんな言語を勉強すべきかを述べている。まず Python と Java から始めよ、学ぶのが容易だから。真剣なハッカーはさらに、Unix をハックするために C を学び、システム管理と CGI スクリプトのために Perl を学ぶべし。そして本当に真剣なハッカーは Lisp を学ぶことを熟慮すべきだ、と。

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

Lisp は、それをものにした時の素晴らしい悟り体験のために勉強しましょう。この体験は、その後の人生でより良いプログラマとなる手助けとなるはずです。たとえ、実際には Lisp そのものをあまり使わなくても。

This is the same argument you tend to hear for learning Latin. It won't get you a job, except perhaps as a classics professor, but it will improve your mind, and make you a better writer in languages you do want to use, like English.

これはまるでラテン語の勉強について語っているみたいじゃないか。ラテン語を勉強しても就職には役に立たない(まあ、古典の教授を除いては)。でもそれはあなたの心を豊かにし、英語なり何んりの実際に使う言葉をよりうまく使えるようになる。

But wait a minute. This metaphor doesn't stretch that far. The reason Latin won't get you a job is that no one speaks it. If you write in Latin, no one can understand you. But Lisp is a computer language, and computers speak whatever language you, the programmer, tell them to.

でもちょっと待った。このメタファーをそこまで敷衍していいのか。ラテン語を勉強しても職が無いのは誰もそれを話さないからだ。ラテン語で何か書いたって、誰も読んじやくれない。でも Lisp はコンピュータ言語だ。コンピュータは、何であろうとプログラマが選

んだ言語を話すんじゃないか。

So if Lisp makes you a better programmer, like he says, why wouldn't you want to use it? If a painter were offered a brush that would make him a better painter, it seems to me that he would want to use it in all his paintings, wouldn't he? I'm not trying to make fun of Eric Raymond here. On the whole, his advice is good. What he says about Lisp is pretty much the conventional wisdom. But there is a contradiction in the conventional wisdom: Lisp will make you a better programmer, and yet you won't use it.

だとしたら、もしエリックが言うように Lisp が良いプログラマーを作るなら、Lisp を使ってみちゃあどうか。画家が、これを使えば腕が上がるという筆を手にしたら、自分の描く絵でそれを使ってみたいだろうと私は思うんだ。別に私はここでエリックの挙げ足を取っているわけじゃない。全体として、彼のアドバイスはとても良い。彼が Lisp について言っていることは世間一般の通念だ。つまり、Lisp を学べばよいプログラマーになれる、でもそれを実際に使うことはない、と。

Why not? Programming languages are just tools, after all. If Lisp really does yield better programs, you should use it. And if it doesn't, then who needs it?

なぜだい? プログラミング言語なんてただの道具じゃないか。Lisp で良いプログラムが書けるなら、使うべきなんだ。そうでないならいったい何の役に立つっていうんだい?

This is not just a theoretical question. Software is a very competitive business, prone to natural monopolies. A company that gets software written faster and better will, all other things being equal, put its competitors out of business. And when you're starting a startup, you feel this very keenly. Startups tend to be an all or nothing proposition. You either get rich, or you get nothing. In a startup, if you bet on the wrong technology, your competitors will crush you.

これはただの机上の空論じゃない。ソフトウェアビジネスは極めて競争の激しい業界で、結果的に自然な独占を許す傾向にある。ある会社が他の会社より、他が同じ条件で、より良いソフトウェアをより速く書いたとしたら、他の会社はいずれ潰れる。ベンチャー企業では特にそうだ。ベンチャーは、オール・オア・ナッシングの勝負になりがちだ。大金持ちになるか、何も手に入らないか。ベンチャー企業では、もし間違った技術に賭けてしまったら、いずれ競争相手に潰されるだろう。

Robert and I both knew Lisp well, and we couldn't see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software

in C++ or Perl. But we also knew that that didn't mean anything. If you chose technology that way, you'd be running Windows. When you choose technology, you have to ignore what other people are doing, and consider only what will work the best.

ロバートと私はともによく Lisp を知っていて、私たちの直観を覆すようないかなる理由も見当たらなかった。他の誰もが C++ や Perl を使っていることは知っていた。でもそれは私たちには何の意味もないことだった。もし、他の誰もが使っているからという理由で技術を選ぶなら、Windows を走らせておけばいいのさ。技術を選択する時は、他の人がどうやっているかなんて無視して、何が最適かを見極めることだけを考えるべきだ。

This is especially true in a startup. In a big company, you can do what all the other big companies are doing. But a startup can't do what all the other startups do. I don't think a lot of people realize this, even in startups.

特にベンチャー企業ではそうだ。大企業では、他の大企業がやっているのと同じようなことをしていてもよい。ベンチャーは他のベンチャーと同じことをやっていてはいけない。このことに、ベンチャー企業の中にいてさえ気付いていない人が多いんじゃないかと思う。

The average big company grows at about ten percent a year. So if you're running a big company and you do everything the way the average big company does it, you can expect to do as well as the average big company-- that is, to grow about ten percent a year.

平均的な大企業は年に 10% 成長する。だから、あなたが大企業を経営していて、他の大企業がやっているのと全く同じようにやっていけば、やはり年に 10% くらいの成長が期待できるだろう。

The same thing will happen if you're running a startup, of course. If you do everything the way the average startup does it, you should expect average performance. The problem here is, average performance means that you'll go out of business. The survival rate for startups is way less than fifty percent. So if you're running a startup, you had better be doing something odd. If not, you're in trouble.

もちろん、同じ論理はベンチャー企業にも成り立つ。もしあなたが平均的なベンチャー企業と同じことをやっていけば、平均的な成長率が期待できる。問題は、だ。ベンチャー企業の平均的な成長とは、すなわち、潰れてしまうということだ。ベンチャー企業の生存率は 50% よりはるかに小さい。したがって、ベンチャーをやるなら何か変わったことをしなければならない。そうでなければ、困ったことになる。

Back in 1995, we knew something that I don't think our competitors understood, and few understand even now: when you're writing software that only has to run on your own servers, you can use any language you want. When you're writing desktop software, there's a strong bias toward writing applications in the same language as the operating system. Ten years ago, writing applications meant writing applications in C. But with Web-based software, especially when you have the source code of both the language and the operating system, you can use whatever language you want.

1995 年当時、私たちは、たぶん競争相手は理解してないであろうあることを知っていた。いや、今でも理解している人はほとんどいないかもしれない。もしあなたが、あなたのサーバ上でだけ走るソフトウェアを書くのなら、あなたは自分の好むどんな言語でも使えるということだ。デスクトップソフトを書いている時にはこうはいかない。そのデスクトップのオペレーティングシステムがサポートする言語への強いバイアスがある。10 年前なら、アプリケーションを書くということは C で書くということだった。でも Web ベースのソフトウェアなら、そして特に言語と OS のソースコードを持っているなら、あなたは好きな言語を使うことができる。

This new freedom is a double-edged sword, however. Now that you can use any language, you have to think about which one to use. Companies that try to pretend nothing has changed risk finding that their competitors do not.

この新しい自由は、しかし、両刃の剣でもある。どんな言語でも使えるということは、どれを使うかを考えなくちゃならないということだ。あなたがその違いに気付かない振りをしていたら、競争相手に差をつけられてしまうリスクがある。

If you can use any language, which do you use? We chose Lisp. For one thing, it was obvious that rapid development would be important in this market. We were all starting from scratch, so a company that could get new features done before its competitors would have a big advantage. We knew Lisp was a really good language for writing software quickly, and server-based applications magnify the effect of rapid development, because you can release software the minute it's done.

もしどんな言語でも使えるとなったら、いったい何を使う? 私たちは Lisp を選んだ。ひとつには、このマーケットでは素早く開発することが重要だというのが明らかだったからだ。私たちも競争相手もゼロから始めるところで、だから競争相手より少しでも速く新しい機能を実現できればそれは大きなリードになった。Lisp はソフトウェアを素早く書くのに良

い言語だ。そしてサーバベースのアプリケーションはその利点を増幅する。完成したらすぐにリリースできるからだ。

If other companies didn't want to use Lisp, so much the better. It might give us a technological edge, and we needed all the help we could get. When we started Viaweb, we had no experience in business. We didn't know anything about marketing, or hiring people, or raising money, or getting customers. Neither of us had ever even had what you would call a real job. The only thing we were good at was writing software. We hoped that would save us. Any advantage we could get in the software department, we would take.

他の会社が Lisp を使いたがらなければ、それは良いことだった。それは私たちにとって技術的に有利な点になり得たし、スタートしたばかりのベンチャー企業としては何であれ有利な材料は必要だった。Viaweb を始めたとき、私たちにはビジネスの経験がなかった。マーケティングのことも、人を雇うことも、資金を集めることも、顧客を得ることも、何ひとつ知らなかった。そして私たちのどちらも、あなたが「現実の仕事」と呼ぶものに就いたことさえなかった。たったひとつ、私たちがよくできたのはソフトウェアを書くことだった。私たちはそれに賭けた。ソフトウェア部門に有利になることなら何だって採り入れようとした。

So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better product for less money, and still make a profit. We would end up getting all the users, and our competitors would get none, and eventually go out of business. That was what we hoped would happen, anyway.

これを聞いたら、Lisp を使ったことは実験だったって思うかもしれないね。私たちの仮説は、Lisp でソフトウェアを書けば、競争相手より速く機能を実装することができて、また競合ソフトウェアができないことができるようになるだろうというものだった。そして Lisp は非常に高機能なため、大きな開発チームは必要なくて、開発費用も抑えられるだろう。そうなればより安く、より良い製品を提供することができて、利益を上げられるだろう。全てのユーザをこっちに引き付け、競争相手を蹴散らせるだろう。とにかくこれが私たちが目論んだことだった。

What were the results of this experiment? Somewhat surprisingly, it worked. We

eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt like a desktop application. Our competitors had cgi scripts. And we were always far ahead of them in features. Sometimes, in desperation, competitors would try to introduce features that we didn't have. But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too.

実験の結果はどうだったかって?ちょっとびっくりすることに、うまくいったんだ。結果的に私たちには 20 から 30 にものぼる競争相手が現われたが、どれひとつとして私たちのソフトウェアを打ち負かすことはできなかった。私たちの WYSIWYG のオンラインストアビルダーはサーバ側で走っているのに、ユーザからはまるでデスクトップアプリケーションを使っているかのように感じられた。競争相手は CGI スクリプトを使っていた。そして機能的にも、私たちは相手のはるか先を行っていた。しばしば、競争相手は追い詰められて、私たちのソフトにない機能を入れようと試みた。しかし、Lisp のおかげで私たちの開発サイクルは非常に速く、相手がプレスリリースを出した 1 日 2 日後にもう同様の機能を作ったこともしばしばある。プレスリリースをカバーした記者が私たちのところに取材に来るころには、こちらにもその新しい機能が追加されていたのだ。

It must have seemed to our competitors that we had some kind of secret weapon-- that we were decoding their Enigma traffic or something. In fact we did have a secret weapon, but it was simpler than they realized. No one was leaking news of their features to us. We were just able to develop software faster than anyone thought possible.

きっと競争相手には、私たちが何か秘密兵器を持っているかのように見えたに違いない。彼らの暗号通信を解読しているとか、そんなことだ。事実、私たちは秘密兵器を持っていたんだが、それは彼らが思っているよりずっと簡単なことだった。誰も彼らの計画を私たちに漏らしたりしなかった。単に私たちが、誰も想像できないくらい素早くソフトウェアを開発できたというだけのことだ。

When I was about nine I happened to get hold of a copy of *The Day of the Jackal*, by Frederick Forsyth. The main character is an assassin who is hired to kill the president of France. The assassin has to get past the police to get up to an apartment that overlooks the president's route. He walks right by them, dressed up as an old man on crutches, and they never suspect him.

9 歳くらいの時、私はフレデリック・フォーサイスの「ジャッカルの日」を読んだ。主人公はフランスの大統領を殺すために雇われた暗殺者だ。彼は大統領の通る道を見下ろすアパートに登るために警官の警備網を抜けなければならなかったのだが、松葉杖をついた老人の格好をするだけで、やすやすと警官の前を疑われずに通り抜けたのだ。

Our secret weapon was similar. We wrote our software in a weird AI language, with a bizarre syntax full of parentheses. For years it had annoyed me to hear Lisp described that way. But now it worked to our advantage. In business, there is nothing more valuable than a technical advantage your competitors don't understand. In business, as in war, surprise is worth as much as force.

私たちの秘密兵器も同じようなものだ。私たちはソフトウェアを、括弧だらけの奇怪な構文を持つ、妙ちきりんな人工知能言語で書いた。正直なところ何年もの間、私は Lisp がそんなふうに呼ばれることが気に障っていたのだが、いまやそれは私たちのアドバンテージとなったのだ。ビジネスでは、競争相手が理解出来ない技術的アドバンテージを持っていることほど価値あることはない。ビジネスでは、驚きは軍隊ほどに価値がある。

And so, I'm a little embarrassed to say, I never said anything publicly about Lisp while we were working on Viaweb. We never mentioned it to the press, and if you searched for Lisp on our Web site, all you'd find were the titles of two books in my bio. This was no accident. A startup should give its competitors as little information as possible. If they didn't know what language our software was written in, or didn't care, I wanted to keep it that way. [2]

そして、ちょっと恥を忍んで告白すると、私たちが Viaweb を作っている間、私は Lisp について一切公言しなかった。プレス発表でも言わなかったし、私たちの Web サイトで「Lisp」を検索しても見つかるのは私の書いた Lisp に関する 2 冊の書物のタイトルだけだ。これは偶然そうだったわけじゃない。ベンチャー企業はライバルになるべく情報を漏らさないものだ。もしライバルが、私たちのソフトが何で書かれているか知らないのなら、あるいは気にしないのなら、私はそれをそのままにしておきたかった。[2]

The people who understood our technology best were the customers. They didn't care what language Viaweb was written in either, but they noticed that it worked really well. It let them build great looking online stores literally in minutes. And so, by word of mouth mostly, we got more and more users. By the end of 1996 we had about 70 stores online. At the end of 1997 we had 500. Six months later, when Yahoo bought us, we had 1070 users. Today, as Yahoo Store, this software continues to dominate its market. It's one of the more profitable pieces of Yahoo,

and the stores built with it are the foundation of Yahoo Shopping. I left Yahoo in 1999, so I don't know exactly how many users they have now, but the last I heard there were about 20,000.

私たちの技術を一番理解していたのは私たちの顧客だった。お客さんは **Viaweb** が何の言語で書かれているかなんて気にしないが、ソフトがとにかく使えるということに気付いてくれた。私たちのソフトを使えば、見栄えがいいオンラインストアを文字通り数分のうちに作ることができた。そしてほとんどロコミによって私たちは顧客を獲得していった。1996 年の終りまでに、70 店ほどがオンラインストアを実現した。1997 年の終りには 500 店になった。その 6 ヶ月後、**Yahoo** が私たちの会社を買い取った時、顧客数は 1070 店になっていた。今日、私たちのソフトウェアは **Yahoo Store** としてこの領域のマーケットを独占している。それは **Yahoo** の中で最も利益を上げている部分のひとつであり、それによって作られたオンラインストアが **Yahoo Shopping** の基礎となっている。私は 1999 年に **Yahoo** を離れたので、現在どのくらいのユーザがいるのかは知らないが、最後に聞いた時は 2 万以上ということだった。

The Blub Paradox

「ほげ言語」のパラドックス

What's so great about Lisp? And if Lisp is so great, why doesn't everyone use it? These sound like rhetorical questions, but actually they have straightforward answers. Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available. And the reason everyone doesn't use it is that programming languages are not merely technologies, but habits of mind as well, and nothing changes slower. Of course, both these answers need explaining.

Lisp の何がそんなに素晴らしいんだ?それに、もし Lisp がそんなに良いのなら、どうしてみんなそれを使わないんだ?これは修辞疑問みたいに聞こえるが、きちんとした答えのある質問だ。Lisp は、信者のみが見ることができる魔法の特性があるから素晴らしいんじゃない。単に、今ある言語の中でもっともパワフルだからだ。そして、みんながそれを使わない理由は、プログラミング言語とは単なる技術だけでなく、心の習慣でもあり、それは最も変化の遅いものであるというものだ。もちろん、この 2 つの答えはもっとよく説明されなければなるまい。

I'll begin with a shockingly controversial statement: programming languages vary in power.

まず、思いつきり物議を醸しそうな発言から始めよう。プログラミング言語は、その力に

おいて差がある。

Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you. This idea is even built into the hardware now: since the 1980s, instruction sets have been designed for compilers rather than human programmers.

少なくとも、高級言語は機械語より力があるということに反対する人はほとんどいないだろう。今日ではほとんどのプログラマは、通常、機械語でプログラムを書きたいとは思わないということに同意するだろう。代わりに高級言語で書いて、コンパイラにそれを機械語に変換させるべきなのだ。この考えは既にハードウェアの設計に採り入れられている。1980年代以降、命令セットは人間のプログラマよりはコンパイラ向けに設計されるようになった。

Everyone knows it's a mistake to write your whole program by hand in machine language. What's less often understood is that there is a more general principle here: that if you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one. [3]

あなたがプログラムの全てを機械語で書くと言い出したら、誰もがそれは間違いだと言うだろう。しかし、同じ原理だが見過ごされがちな一般法則がある。もしあなたがいくつかの言語を選択することができて、他の条件が同じ場合、最も力のある言語以外を選択することは間違いなのだ。[3]

There are many exceptions to this rule. If you're writing a program that has to work very closely with a program written in a certain language, it might be a good idea to write the new program in the same language. If you're writing a program that only has to do something very simple, like number crunching or bit manipulation, you may as well use a less abstract language, especially since it may be slightly faster. And if you're writing a short, throwaway program, you may be better off just using whatever language has the best library functions for the task. But in general, for application software, you want to be using the most powerful (reasonably efficient) language you can get, and using anything else is a mistake, of exactly the same kind, though possibly in a lesser degree, as programming in machine language.

この法則には例外がたくさんある。例えばあなたの書くプログラムが、別の、特定の言語

で書かれたプログラムと緊密に動作しなければならないとしたら、たぶん同じ言語でプログラムを書くのが良い方策だろう。もしあなたのプログラムが非常に簡単なこと、たくさん
の計算をこなしたりビット列を処理したりするなら、あまり抽象度の高い言語でないほうがいい。抽象度の低い言語の方がそういう動作が速いからだ。そしてもしあなたが、小さい、使い捨てのプログラムを書いているなら、目的に一番合ったライブラリを持っている言語を使うのが良いだろう。でも一般的には、アプリケーションを書くなら、手に入るなかで最も力の強い(そしてそれなりに効率の良い)言語を使うべきだし、そうでない選択は、程度の差こそあれ、そこで機械語を使うことが間違いであるのと同様の理由で、間違いなのだ。

You can see that machine language is very low level. But, at least as a kind of social convention, high-level languages are often all treated as equivalent. They're not. Technically the term "high-level language" doesn't mean anything very definite. There's no dividing line with machine languages on one side and all the high-level languages on the other. Languages fall along a continuum [4] of abstractness, from the most powerful all the way down to machine languages, which themselves vary in power.

機械語は非常に低レベルだということは誰にでもわかる。しかし、少なくとも社会的な慣習として、高級言語は対等であるとされることが多い。そんなわけではない。「高級言語」なんて用語は何も定義しない。一方に機械語を、もう一方に他の全ての高級言語を隔てる線なんてありはしない。プログラミング言語はそれぞれが、機械語から最も力のある言語までの連続した抽象度のスペクトルのどこかに位置するのだ。[4]

Consider Cobol. Cobol is a high-level language, in the sense that it gets compiled into machine language. Would anyone seriously argue that Cobol is equivalent in power to, say, Python? It's probably closer to machine language than Python.

Cobol を考えてみたまえ。Cobol は、それがコンパイルされて機械語に落とされるという点では高級言語だ。しかし、Cobol が例えば Python と同じ力を持つ、と真剣に主張した人はいるか? たぶん Cobol は Python よりは機械語に近い。

Or how about Perl 4? Between Perl 4 and Perl 5, lexical closures got added to the language. Most Perl hackers would agree that Perl 5 is more powerful than Perl 4. But once you've admitted that, you've admitted that one high level language can be more powerful than another. And it follows inexorably that, except in special cases, you ought to use the most powerful you can get.

じゃあ Perl 4 はどうだろう。Perl 4 から Perl 5 になる時に、レキシカルクロージャが追加

された。多くの Perl ハッカーは、Perl 5 は Perl 4 よりパワフルだということに同意するだろう。しかしそれを認めるということは、高級言語の間に力の差があるということを確認することになる。そしてそれを認めることは、特別な場合を除いては、最もパワフルな言語を使うべきってことになる。

This idea is rarely followed to its conclusion, though. After a certain age, programmers rarely switch languages voluntarily. Whatever language people happen to be used to, they tend to consider just good enough.

でも、この考えが結論のところまでたどられることは滅多にない。ある年齢に達すると、プログラマは自分から使う言語を変えることはほとんどなくなる。どの言語を使っていようと、これで十分だと思ってしまうのだ。

Programmers get very attached to their favorite languages, and I don't want to hurt anyone's feelings, so to explain this point I'm going to use a hypothetical language called Blub. Blub falls right in the middle of the abstractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language.

プログラマは自分の好みの言語を深く愛する質だし、私は誰も傷つけないので、ここで仮想的なプログラミング言語「ほげ」を使って私のポイントを説明しよう。「ほげ」は抽象度のスペクトルのちょうど真ん中に位置するものとする。最もパワフルな言語ではないけれど、Cobol や機械語よりはパワフルだ。

And in fact, our hypothetical Blub programmer wouldn't use either of them. Of course he wouldn't program in machine language. That's what compilers are for. And as for Cobol, he doesn't know how anyone can get anything done with it. It doesn't even have x (Blub feature of your choice).

そして、仮想的な「ほげ」プログラマ氏は、Cobol も機械語も使わない。もちろん機械語なんて使わない。あれはコンパイラのためのものだ。それに Cobol だって、あれで何かをきちんと動かしたことがあるって人を知らないよ。結局のところ、「ほげ」にある機能 x がないもんな。

As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird

languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub.

このプログラマ氏がパワーのスペクトルを見下ろしているとき、彼にはそうしているという自覚がある。「ほげ」よりも力の弱い言語は、明らかに力が弱い。彼が慣れ親しんだ機能がないからだ。しかし、このプログラマ氏が反対の方向に目を転じた時、彼は自分が見上げているのだということに気付かないのだ。彼が目にするのは、変てこりんな言語ばかり。たぶん、それらは「ほげ」と同じくらいパワフルなんだろうけど、どういうわけかふわふわしたおまけがいろいろ付いているんだ、と思うだろう。彼にとっては「ほげ」で十分なのだ。なぜなら彼は「ほげ」で考えているから。

When we switch to the point of view of a programmer using any of the languages higher up the power continuum, however, we find that he in turn looks down upon Blub. How can you get anything done in Blub? It doesn't even have y.

でも、これよりもう一段階パワフルな言語を使っているプログラマを考えてみると、たぶん彼は「ほげ」を見下ろす位置にいる。いったい「ほげ」で何ができるっていうんだい。あれには機能 y さえないじゃないか。

By induction, the only programmers in a position to see all the differences in power between the various languages are those who understand the most powerful one. (This is probably what Eric Raymond meant about Lisp making you a better programmer.) You can't trust the opinions of the others, because of the Blub paradox: they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.

帰納的に、全てのプログラミング言語の力の違いを見分けることができるのは、最もパワフルな言語を理解するプログラマのみであるということになる(これが、エリックが **Lisp** は良いプログラマを作ると言った真意だろうと思う)。この「ほげ」のパラドックスのために、他の人の意見は参考にならない。誰であれ、プログラムについて考えるときには自分が使うことになった言語に思考が支配されているから、その言語に満足してしまうんだ。

I know this from my own experience, as a high school kid writing programs in Basic. That language didn't even support recursion. It's hard to imagine writing programs without using recursion, but I didn't miss it at the time. I thought in Basic. And I was a whiz at it. Master of all I surveyed.

私は自分の経験からこのことを知った。高校生の時私は **Basic** でプログラムを書いていたんだ。再帰もない言語でだ。再帰なしでプログラムを書くなんて今じゃ考えられないけれ

ど、当時は何とも思わなかった。全てを **Basic** で考えていたからだ。そして私は **Basic** の達人だった。知る限り全てのことをマスターしていた。

The five languages that Eric Raymond recommends to hackers fall at various points on the power continuum. Where they fall relative to one another is a sensitive topic. What I will say is that I think Lisp is at the top. And to support this claim I'll tell you about one of the things I find missing when I look at the other four languages. How can you get anything done in them, I think, without macros?

[\[5\]](#)

エリックがハッカーに薦めた 5 つの言語は、パワースペクトルの違った場所にそれぞれ位置する。どれがどこに来るかは微妙な問題だけれど、私は、**Lisp** は一番上に来ると言おう。そしてこの主張を裏付けるために、他の 4 つの言語に欠けているあるひとつの機能を言おう。いったいこの機能なしでどうやってプログラムを書くんだい?と私が考えるのが、マクロだ。 [\[5\]](#)

Many languages have something called a macro. But Lisp macros are unique. And believe it or not, what they do is related to the parentheses. The designers of Lisp didn't put all those parentheses in the language just to be different. To the Blub programmer, Lisp code looks weird. But those parentheses are there for a reason. They are the outward evidence of a fundamental difference between Lisp and other languages.

たくさんの言語が、マクロと呼ばれる機能を持っている。でも **Lisp** のマクロは特別だ。そして、信じられないかもしれないけれど、それが強力なのは **Lisp** のあのたくさんの括弧と深い関係がある。**Lisp** は別に他と違いを出すために括弧だらけになったんじゃない。「ほげ」プログラマにしたら、**Lisp** の括弧は変てこりにしか見えないだろうけど、あれにはちゃんと理由があるんだ。そしてそれが、**Lisp** と他の言語との根本的な違いの証拠なんだ。

Lisp code is made out of Lisp data objects. And not in the trivial sense that the source files contain characters, and strings are one of the data types supported by the language. Lisp code, after it's read by the parser, is made of data structures that you can traverse.

Lisp のプログラムコードは **Lisp** のデータオブジェクトからできている。それは、ソースコードは文字列でできていて、文字列は言語でサポートされている、というようなつまらない意味じゃない。**Lisp** のコードは、ひとたびパーザによって読まれたら、あなたが解析することができるデータ構造になるんだ。

If you understand how compilers work, what's really going on is not so much that Lisp has a strange syntax as that Lisp has no syntax. You write programs in the parse trees that get generated within the compiler when other languages are parsed. But these parse trees are fully accessible to your programs. You can write programs that manipulate them. In Lisp, these programs are called macros. They are programs that write programs.

コンパイラの動作を理解していれば、Lisp の構文は奇妙だと言っても Lisp には構文がないと言っても大した違いはないということが分かるだろう。他の言語ならコンパイラが構文解析して内部に作られる構文木を、Lisp では直接プログラムとして書き下すわけだ。しかも、この構文木はプログラムからアクセスできるから、構文木自身を操作するプログラムを書くことができる。Lisp ではそのようなプログラムをマクロと呼ぶ。いわば、プログラムを生成するプログラムだ。

Programs that write programs? When would you ever want to do that? Not very often, if you think in Cobol. All the time, if you think in Lisp. It would be convenient here if I could give an example of a powerful macro, and say there! how about that? But if I did, it would just look like gibberish to someone who didn't know Lisp; there isn't room here to explain everything you'd need to know to understand what it meant. In [Ansi Common Lisp](#) I tried to move things along as fast as I could, and even so I didn't get to macros until page 160.

プログラムを生成するプログラムだって?いつそんなものが必要なんだ?そんなこと滅多にないよ、と Cobol プログラマは言うだろう。いつでもさ、と Lisp で考えるプログラマは言うだろう。ここでパワフルなマクロの例をここで見せて、ほらどうだい、と言えればいいんだろうけれど、そうしたところできっと Lisp を知らない人にはわけのわからないデタラメ語にしか見えないだろう。その例が何を意味するかを説明しているだけのスペースはここにはない。「ANSI Common Lisp」を書いた時、私はなるべく素早くいろいろなことを詰め込もうとしたのだけれど、それでもマクロは 160 ページになるまで出せなかった。

But I think I can give a kind of argument that might be convincing. The source code of the Viaweb editor was probably about 20-25% macros. Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary. So every macro in that code is there because it has to be. What that means is that at least 20-25% of the code in this program is doing things that you can't easily do in any other language. However skeptical the Blub programmer might be about my claims for the mysterious powers of Lisp, this ought to make him curious. We weren't writing this code for our own

amusement. We were a tiny startup, programming as hard as we could in order to put technical barriers between us and our competitors.

でも別の数字を使って多少説得力のある議論は出来るかもしれない。Viaweb エディタのソースコードのうち 20~25% はマクロだ。マクロは通常の Lisp 関数より書くのが大変だし、必要のないところで使うのは良くないスタイルとされている。だから、このプログラムコードにあるマクロは全て必要があって書かれたんだ。つまり、このプログラムの、少なくとも 20~25% の部分は、他の言語では簡単にできないような真似をやってのけているってことだ。「ほげ」プログラマが私の主張する Lisp の神秘的な力にどんなに懐疑的であっても、この事実にはちょっと好奇心をくすぐられるんじゃないか。別に私たちは自分の楽しみのためにこれらのコードを書いていたんじゃない。私たちは小さなベンチャーで、私たちとライバルとの間の技術的な差を広げるために可能な限り、がりがりとコーディングしていたんだ。

A suspicious person might begin to wonder if there was some correlation here. A big chunk of our code was doing things that are very hard to do in other languages. The resulting software did things our competitors' software couldn't do. Maybe there was some kind of connection. I encourage you to follow that thread. There may be more to that old man hobbling along on his crutches than meets the eye. 勘の良い人は、ここに何かの関係があるんじゃないかと思い始めるだろう。私たちのコードの大きな部分は、他の言語を使ったらものすごく難しいようなことをやっていた。出来上がったソフトウェアは、ライバルのソフトウェアにはできないようなことができた。おそらくこれには関係があるんだ。このことを深く考えてみてほしい。あの老人が杖をついていたということには、目に見えること以上に深い意味があるんだ。

Aikido for Startups

ベンチャーのための合気道

But I don't expect to convince anyone ([over 25](#)) to go out and learn Lisp. The purpose of this article is not to change anyone's mind, but to reassure people already interested in using Lisp-- people who know that Lisp is a powerful language, but worry because it isn't widely used. In a competitive situation, that's an advantage. Lisp's power is multiplied by the fact that your competitors don't get it.

でも私は、(25 歳過ぎの)誰かの心を変えて今から Lisp を勉強させることができるとは思っていない。この記事の目的は人の心を変えようということじゃなく、既に Lisp に興味を持っている人の後押しをしようということなんだ。Lisp はパワフルな言語だとは知っている

けど、あんまり使われていないから不安だなあという人たちだ。競争の激しい状況では、そのことは有利になるんだ。Lisp の力は、ライバルがそれを理解できないということによって増幅されるんだ。

If you think of using Lisp in a startup, you shouldn't worry that it isn't widely understood. You should hope that it stays that way. And it's likely to. It's the nature of programming languages to make most people satisfied with whatever they currently use. Computer hardware changes so much faster than personal habits that programming practice is usually ten to twenty years behind the processor. At places like MIT they were writing programs in high-level languages in the early 1960s, but many companies continued to write code in machine language well into the 1980s. I bet a lot of people continued to write machine language until the processor, like a bartender eager to close up and go home, finally kicked them out by switching to a risc instruction set.

もしあなたがベンチャーを立ち上げようとしていて、Lisp を使おうかと考えているなら、他で使われていないからなんて心配してちゃいけない。今後も他で使われないことを期待しなきゃ。そしておそらくその期待は正しい。今使っているプログラミング言語で満足してしまうようにするってのが、プログラミング言語に元来備わった性質なんだから。コンピュータのハードウェアは人間の習慣よりもはるかに速く変化するから、プログラミングスタイルはプロセッサより 10 年から 20 年は遅れている。MIT のような場所では 1960 年代の始めから高級言語が使われていたけれど、多くの企業では 1980 年代も半ばになるまで機械語でのコーディングがまかり通っていた。たぶん、たくさんのプログラマは、プロセッサが RISC インストラクションセットに切り替えて、店仕舞いして早く家に帰りたいバーテンダーに追い出されるようにしてとうとう機械語でのプログラミングから追い出されることになるまで、機械語でコードを書き続けていたんじゃないかと思うよ。

Ordinarily technology changes fast. But programming languages are different: programming languages are not just technology, but what programmers think in. They're half technology and half religion. [6] And so the median language, meaning whatever language the median programmer uses, moves as slow as an iceberg. Garbage collection, introduced by Lisp in about 1960, is now widely considered to be a good thing. Runtime typing, ditto, is growing in popularity. Lexical closures, introduced by Lisp in the early 1970s, are now, just barely, on the radar screen. Macros, introduced by Lisp in the mid 1960s, are still terra incognita.

普通の技術は速く変化する。でもプログラミング言語はちょっと違う。プログラミング言

語は単なる技術ではなく、プログラマがそれを道具として思考するものだからだ。プログラミング言語は半分技術で、半分は宗教なんだ。[\[6\]](#) だから中くらいのプログラミング言語、つまり普通のプログラマが使うような言語ということだが、それはまるで氷山の動きのようにゆっくりと変化する。1960 年頃に Lisp によって導入されたガベージコレクションは、近年では良い技術だと広く認められるようになった。動的型付けも同じくポピュラーになりつつある。レキシカルクロージャは 1970 年代初めに Lisp によって導入されたが、ようやくレーダーの端に捉えられ始めた。1960 年代中頃に Lisp が導入したマクロは、まだ未知の世界だ。

Obviously, the median language has enormous momentum. I'm not proposing that you can fight this powerful force. What I'm proposing is exactly the opposite: that, like a practitioner of Aikido, you can use it against your opponents.

もちろん、普通の言語は恐るべき慣性を持っている。この力と闘えなんて言うつもりはない。むしろ私が言いたいのは全く逆のことなんだ。つまり、合気道家がやるように、相手の力を相手を倒すのに利用するんだ。

If you work for a big company, this may not be easy. You will have a hard time convincing the pointy-haired boss to let you build things in Lisp, when he has just read in the paper that some other language is poised, like Ada was twenty years ago, to take over the world. But if you work for a startup that doesn't have pointy-haired bosses yet, you can, like we did, turn the Blub paradox to your advantage: you can use technology that your competitors, glued immovably to the median language, will never be able to match.

あなたが大企業で働いているならこれはちょっと難しいかもしれない。つい今しがた、ある言語が世界を支配するみたいな記事(20 年前の Ada がそうだった)を読んだばかりの、髪のとんがった上司を説得して、Lisp で何かを作らせてもらうのはかなり厄介だ。でもあなたが、まだ髪のとんがった上司なんかいないベンチャーで働いているなら、私たちがやったように、「ほげ」のパラドックスを自分に有利なように利用できる。競争相手が普通の言語に張りついて動けないでいるうちに、彼らが決して対抗できない技術を使えるのだ。

If you ever do find yourself working for a startup, here's a handy tip for evaluating competitors. Read their job listings. Everything else on their site may be stock photos or the prose equivalent, but the job listings have to be specific about what they want, or they'll get the wrong candidates.

もしあなたがベンチャーで働いているなら、競争相手の実力を見積もる便利なヒントをあげよう。人材募集内容を見るんだ。彼らのサイトの他のページは見栄えのいい写真や美辞

麗句で彩られているだろうけど、募集内容だけは彼等が一番欲しているものを的確に表現しているはずだ。でなければ見当違いの人しか応募して来ないからね。

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to. The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening-- that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried.

Viaweb で働いていた期間、私は人材募集記事をたくさん見た。新しい競争相手は毎月のように現われた。いつも最初に私がしたことは、まずオンラインデモがあるかどうかをチェックして、それから人材募集を見ることだった。1~2年したら、警戒すべきライバルとそうでないところとを見分けられるようになった。人材募集がいかにも IT といった匂いを漂わせていればいるほど、その企業は脅威ではない。一番安全なのは Oracle の経験者を募集しているところだ。そういうところを警戒する必要は全くない。また、Java や C++ プログラマを募集しているところも安全だ。もし Perl や Python プログラマを募集していたら、ちょっと気を付けたほうがいい。その企業の、少なくとも技術部門は本物のハッカーがやっている可能性が高いからだ。もし私が Lisp ハッカーの募集広告を目にしていたら、きっとかなり心配していただろう。

Notes

注釈

[1] Viaweb at first had two parts: the editor, written in Lisp, which people used to build their sites, and the ordering system, written in C, which handled orders. The first version was mostly Lisp, because the ordering system was small. Later we added two more modules, an image generator written in C, and a back-office manager written mostly in Perl.

[1] 当初、Viaweb は 2 つの部分から構成されていた。Lisp で書かれた、ユーザがサイトを構築するエディタと、C で書かれた注文を処理するシステムだ。最初のバージョンでは注文処理システムは小さかったので、ほとんどの部分が Lisp だった。後に私たちはモジュール

ルをもう 2 つ、C で書かれた画像生成システムと、大部分を Perl で書かれたバックオフィスマネージャを追加した。

In January 2003, Yahoo released a new version of the editor written in C++ and Perl. It's hard to say whether the program is no longer written in Lisp, though, because to translate this program into C++ they literally had to write a Lisp interpreter: the source files of all the page-generating templates are still, as far as I know, Lisp code. (See [Greenspun's Tenth Rule](#).)

2003 年 1 月、Yahoo は C++ と Perl で書かれた新しいバージョンのエディタをリリースした。もはやプログラムは Lisp で書かれているとはいい難かったが、元のプログラムを C++ に変換するに当たって、彼らは文字通り、Lisp インタプリタを書かなければならなかった。ページテンプレートを生成するソースは、私の知る限り、依然として Lisp コードだった(グリーンस्पンの第 10 規則(246 ページ)も参照)。

[2] Robert Morris says that I didn't need to be secretive, because even if our competitors had known we were using Lisp, they wouldn't have understood why: "If they were that smart they'd already be programming in Lisp."

[2] ロバート・モリスは、そんなに秘密にする必要はないんじゃないかと言った。もし私たちが Lisp を使っていることをライバルが知っても、その理由を理解することはできないだろうと。「だってそれが理解できるくらいなら、もう Lisp でプログラムしているはずだよ」

[3] All languages are equally powerful in the sense of being Turing equivalent, but that's not the sense of the word programmers care about. (No one wants to program a Turing machine.) The kind of power programmers care about may not be formally definable, but one way to explain it would be to say that it refers to features you could only get in the less powerful language by writing an interpreter for the more powerful language in it. If language A has an operator for removing spaces from strings and language B doesn't, that probably doesn't make A more powerful, because you can probably write a subroutine to do it in B. But if A supports, say, recursion, and B doesn't, that's not likely to be something you can fix by writing library functions.

[3] 全ての言語はチューリング等価であるという意味で同等の力を持つけれど、プログラマが言語の「力」というのはそういう意味じゃない(誰もチューリングマシンでプログラムを書きたいとは思わないよね)。プログラマが気にする「力」とは、明確に定義できるものではないけれど、例えばある機能を実現するのに、力の弱い言語では、その機能を含むより力強い言語のインタプリタを書くことでしか実現できない、というふうに言え

る。例えば言語 **A** が文字列からスペースを除くオペレータを持っていて、言語 **B** がそれを持っていなかったとしよう。これだけで言語 **A** は言語 **B** より力が強いとは言わないだろう。おそらく言語 **B** で同等のことをするサブルーチンが書けるからだ。けれど、例えば **A** は再帰呼び出しをサポートしていて **B** はそうでないとしたら。ちょっとしたライブラリ呼び出しで **B** を **A** に近づけるのは簡単じゃない。

[4] Note to nerds: or possibly a lattice, narrowing toward the top; it's not the shape that matters here but the idea that there is at least a partial order.

[4] こだわる人への注:あるいは、頂上に近付くほど細くなっているラティス(束)と言ってもよいかもしれない。ここで問題にしているのは形状ではなく、少なくともそこには半順序があるということだ。

[5] It is a bit misleading to treat macros as a separate feature. In practice their usefulness is greatly enhanced by other Lisp features like lexical closures and rest parameters.

[5] マクロをひとつの独立した機能と言ってしまうのはちょっと誤解を招くかもしれない。マクロの有用性は、レキシカルクロージャや **rest** 引数というような他の **Lisp** の機能と組み合わせで生まれてくるものだからだ。

[6] As a result, comparisons of programming languages either take the form of religious wars or undergraduate textbooks so determinedly neutral that they're really works of anthropology. People who value their peace, or want tenure, avoid the topic. But the question is only half a religious one; there is something there worth studying, especially if you want to design new languages.

[6] 結果として、プログラミング言語の比較は宗教戦争になるか、中立であろうとするあまりに人類学の研究かと思まがうような学部生用の教科書にしかない。平和を好むか、大学での終身雇用を手に入れたい人は、この話題を避けて通る。でも、宗教的な部分はこの問題の半分だけなんだ。他の部分には研究すべき価値がある。特にあなたが新しい言語を設計したいと思っている時には。

Chapter 13 REVENGE OF THE NERDS

第 13 章 オタク野郎の復讐

May 2002

"We were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp."

- Guy Steele, co-author of the Java spec

私たちは C++ プログラマに倣^{なら}っている。私たちは C++ の多くを、Lisp までであともう少しというところまで持ってくるのができた。

- ガイ・スティー爾、『Java 言語仕様』の共著者

In the software business there is an ongoing struggle between the pointy-headed academics, and another equally formidable force, the pointy-haired bosses. Everyone knows who the pointy-haired boss is, right? I think most people in the technology world not only recognize this cartoon character, but know the actual person in their company that he is modelled upon.

ソフトウェアビジネスでは、頭のとんがった学者たちと、同じくらい手ごわい勢力である髪のとんがった上司とが絶え間ない抗争を繰り広げている。髪のとんがった上司のことはみんな知ってるよね。テクノロジー業界にいる人ならたいていはこのマンガのキャラクターだけでなく、そのモデルになりそうな現実の人間を社内に見つけられると思う。

The pointy-haired boss miraculously combines two qualities that are common by themselves, but rarely seen together: (a) he knows nothing whatsoever about technology, and (b) he has very strong opinions about it.

髪のとんがった上司は二つの性質を持っている。どちらの性質もごく普通に見られるけど、両方とも兼ね備えるとなると奇跡的だ。その性質とは、(a) 技術に関して何も知らない。そして (b) 技術に関して非常に強い意見を持っている、というものだ

Suppose, for example, you need to write a piece of software. The pointy-haired boss has no idea how this software has to work, and can't tell one programming language from another, and yet he knows what language you should write it in. Exactly. He thinks you should write it in Java.

例えば、あなたがあるソフトウェアを書かなければならないとする。髪の毛のついた上司はそのソフトウェアがどう動くべきかなんて全く理解できないし、あのプログラミング言語とこのプログラミング言語の区別もつけられない。でも不思議と、彼はあなたが使わなければならない言語を知っているんだ。その通り。Java を使いなさい、ってこった。

Why does he think this? Let's take a look inside the brain of the pointy-haired boss. What he's thinking is something like this. Java is a standard. I know it must be, because I read about it in the press all the time. Since it is a standard, I won't get in trouble for using it. And that also means there will always be lots of Java programmers, so if the programmers working for me now quit, as programmers working for me mysteriously always do, I can easily replace them.

何で彼はそう考えるんだろう。この髪の毛のついた上司の頭の中を覗いてみよう。きっとこんなふうに考えているんだろう。Java は標準だ。そうに違いない。だって新聞や雑誌にいつでも載っているじゃないか。それが標準なら、それを使っておけば問題は起こるまい。それにいつだって Java プログラマはたくさんいるだろうから、今働いている部下が辞めてしまっても、いや、私の部下は不思議とよく辞めるんだが、それでもすぐ代わりが見つけれられるというものさ。

Well, this doesn't sound that unreasonable. But it's all based on one unspoken assumption, and that assumption turns out to be false. The pointy-haired boss believes that all programming languages are pretty much equivalent. If that were true, he would be right on target. If languages are all equivalent, sure, use whatever language everyone else is using.

ふーん。確かにこれはそんなにおかしな考えとも思えない。ただこの考えはひとつの言外仮定に基づいていて、実はその仮定は間違いなんだ。髪の毛のついた上司は全ての言語はだいたい等しいと信じている。これが真実なら彼は的を射ている。言語が全て等しいなら、他のみんなが使っている言語を使いたまえ。

But all languages are not equivalent, and I think I can prove this to you without even getting into the differences between them. If you asked the pointy-haired boss in 1992 what language software should be written in, he would have answered with as little hesitation as he does today. Software should be written in C++. But if languages are all equivalent, why should the pointy-haired boss's opinion ever change? In fact, why should the developers of Java have even bothered to create a new language?

実際には、言語は等しくないんだ。各言語の違いに立ち入らなくたってそれを証明することができる。1992 年にかの髪のとんがった上司にどの言語でソフトを書くべきか尋ねたら、彼は今日彼がそうしたのと同じように、ためらいなく答えただろう。ソフトウェアは **C++** で書くべきだ。でも全ての言語が等しいなら、なぜ髪のとんがった上司の意見は変わるんだい。だいいち、なぜ **Java** の開発者たちはわざわざ手間をかけて新しい言語を作ったりしたんだろう。

Presumably, if you create a new language, it's because you think it's better in some way than what people already had. And in fact, Gosling makes it clear in the first Java white paper that Java was designed to fix some problems with C++. So there you have it: languages are not all equivalent. If you follow the trail through the pointy-haired boss's brain to Java and then back through Java's history to its origins, you end up holding an idea that contradicts the assumption you started with.

新しい言語を作るってことは、既に他の人がやったやり方よりうまくできると思うからそうするんだろう。実際、ゴスリンは最初の **Java** のホワイトペーパーで、**Java** は **C++** にあるいくつかの問題を解決すべく設計されたとはっきりと述べている。ほらね。言語は全て等価ではない。髪のとんがった上司の頭から **Java** へと、そして **Java** の歴史をその起源へとたどってみれば、最初の仮定と矛盾する考えにぶちあたるというわけさ。

So, who's right? James Gosling, or the pointy-haired boss? Not surprisingly, Gosling is right. Some languages *are* better, for certain problems, than others. And you know, that raises some interesting questions. Java was designed to be better, for certain problems, than C++. What problems? When is Java better and when is C++? Are there situations where other languages are better than either of them?

じゃあどっちが正しいんだろう。ジェームズ・ゴスリンか、髪のとんがった上司か。もちろんゴスリンが正しい。いくつかの言語は、特定の問題においては、他の言語に優る。**Java** は特定の問題において **C++** より良くなるべく設計された。どんな問題に? どんなとき **Java** は **C++** より優れているんだ? この 2 つ以外の言語が優れるような状況ってあるだろうか。

Once you start considering this question, you have opened a real can of worms. If the pointy-haired boss had to think about the problem in its full complexity, it would make his brain explode. As long as he considers all languages equivalent, all he has to do is choose the one that seems to have the most momentum, and since that is more a question of fashion than technology, even he can probably get

the right answer. But if languages vary, he suddenly has to solve two simultaneous equations, trying to find an optimal balance between two things he knows nothing about: the relative suitability of the twenty or so leading languages for the problem he needs to solve, and the odds of finding programmers, libraries, etc. for each. If that's what's on the other side of the door, it is no surprise that the pointy-haired boss doesn't want to open it.

ひとたびこの問題を考え出すと取り返しのつかないことになる。髪のとんがった上司がこの問題をまともに考え出したら、彼の頭脳は爆発してしまうだろう。全ての言語は等しいと考えておけば、一番勢いのある言語を選べばいいだけだ。それは技術よりはむしろ流行の問題なんだから、そうやって選んだものが正しいかもしれないじゃないか。でも言語が等しくないとする、彼は2つの連立方程式を解いて自分が全く知らない2つの事柄の間の最適なバランスを見つけ出さなくちゃならない。20 かそこいらある最新の言語の解くべき問題に対する相対的な適合性と、各言語に対してライブラリやプログラマを見つけられる確率とだ。そんなものがドアの向こうにあるのなら、髪のとんがった上司がドアを開けたがらないのも無理はない。

The disadvantage of believing that all programming languages are equivalent is that it's not true. But the advantage is that it makes your life a lot simpler. And I think that's the main reason the idea is so widespread. It is a *comfortable* idea. 全ての言語が等価だと信じることのデメリットは、それが真実ではないということだ。だがそう信じることには、人生がよっぽど単純になるというメリットがある。私が思うに、これこそがこの考えが広まっている主たる原因だ。心地良い考えなんだ。

We know that Java must be pretty good, because it is the cool, new programming language. Or is it? If you look at the world of programming languages from a distance, it looks like Java is the latest thing. (From far enough away, all you can see is the large, flashing billboard paid for by Sun.) But if you look at this world up close, you find that there are degrees of coolness. Within the hacker subculture, there is another language called Perl that is considered a lot cooler than Java. Slashdot, for example, is generated by Perl. I don't think you would find those guys using Java Server Pages. But there is another, newer language, called Python, whose users tend to look down on Perl, and [more](#) waiting in the wings. Java は素晴らしいはずだ。だってクールで新しい言語だからね。ほんとにそうかい? プログラミング言語の世界からちょっと離れて見てみれば、確かに **Java** が最新のものに見える。(もっとうーんと離れて見たら、**Sun** が出してるでっかいネオンサインしか見えないだろうけどね)。でもこの世界に近づいてよく見てみれば、クールさにもいろいろあることが見

えてくる。ハッカーのサブカルチャーの中では、Perl と呼ばれる言語があって Java よりずっとクールだと考えられている。例えば Slashdot は Perl で生成されている。あれをやっている連中が Java Server Pages を使うとは思えない。そしてさらに Python と呼ばれるもっと新しい言語があって、そのユーザは Perl を見下しがちだ。さらに多くの言語が舞台袖で出番をうかがっている。

If you look at these languages in order, Java, Perl, Python, you notice an interesting pattern. At least, you notice this pattern if you are a Lisp hacker. Each one is progressively more like Lisp. Python copies even features that many Lisp hackers consider to be mistakes. You could translate simple Lisp programs into Python line for line. It's 2002, and programming languages have almost caught up with 1958.

Java、Perl、Python と順に見てくると、面白いパターンに気付くだろう。少なくともあなたが Lisp ハッカーであればきっと気付くはずだ。各言語は次第に Lisp に近付いてきている。Python は Lisp ハッカーの多くが間違いだと思っている機能さえコピーしている。簡単な Lisp プログラムなら、ほぼ一行ごとに Python に変換することが可能だろう。2002 年の今になって、プログラミング言語はとうとう 1958 年に追いつこうとしている。

Catching Up with Math

数学に追いつく

What I mean is that Lisp was first discovered by John McCarthy in 1958, and popular programming languages are only now catching up with the ideas he developed then.

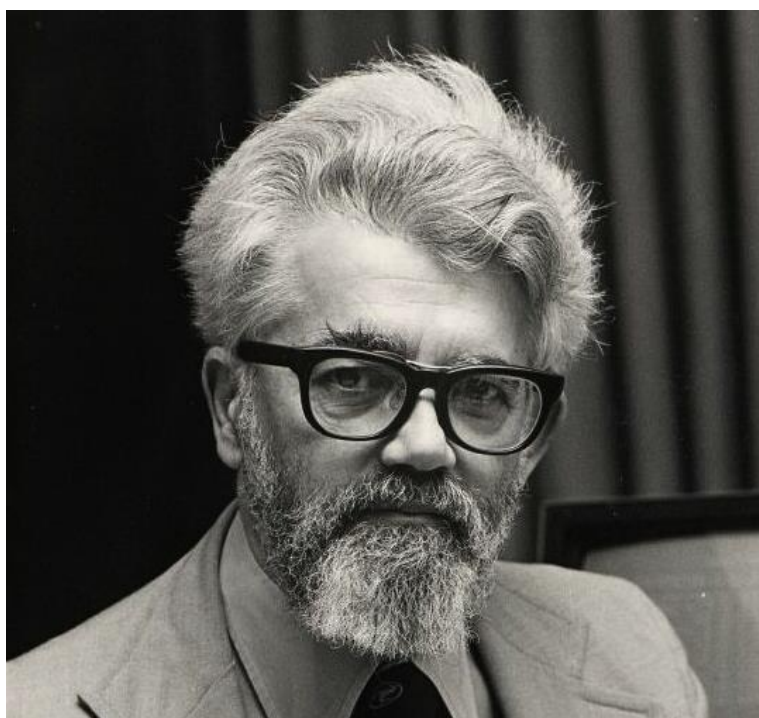
Lisp は 1958 年にジョン・マッカーシーによって最初に発見された。そして、人気のあるプログラミング言語は彼が当時発展させたアイデアにやっと追いついてきた。

Now, how could that be true? Isn't computer technology something that changes very rapidly? I mean, in 1958, computers were refrigerator-sized behemoths with the processing power of a wristwatch. How could any technology that old even be relevant, let alone superior to the latest developments?

こんなことってあり得るだろうか。コンピュータ技術はものすごく速く変化しているんじゃないのか? 1958 年と言えば、コンピュータは冷蔵庫くらい馬鹿でかい化け物で、でも腕時計くらいの計算能力しか持ちちゃいなかった。そんな古い技術が今でも使えるということさえ驚きなのに、最新の開発技術よりも優れているなんてことがあるのだろうか。

I'll tell you how. It's because Lisp was not really designed to be a programming language, at least not in the sense we mean today. What we mean by a programming language is something we use to tell a computer what to do. McCarthy did eventually intend to develop a programming language in this sense, but the Lisp that we actually ended up with was based on something separate that he did as a [theoretical exercise](#)-- an effort to define a more convenient alternative to the Turing Machine. As McCarthy said later,

その秘密はこうだ。Lisp はプログラミング言語として設計されたんじゃない。少なくとも私たちが今日使うプログラミング言語と言う意味では。つまりコンピューターに何をすべきかを指示するもの、という意味ではね。マッカーシーはその後確かにそういう意味でのプログラミング言語を作ろうとしたけど、今日の Lisp になったものは彼が理論的な実験としてやったもの、チューリングマシンのより便利な代替物を定義しようとした試みの結果なんだ。マッカーシーは後にこう語っている。



The greatest nerd, John McCarthy
最高級のオタク、ジョン・マッカーシー

Another way to show that Lisp was neater than Turing machines was to write a universal Lisp function and show that it is briefer and more comprehensible than the description of a universal Turing machine. This was the Lisp function [eval](#)..., which computes the value of a Lisp

expression.... Writing *eval* required inventing a notation representing Lisp functions as Lisp data, and such a notation was devised for the purposes of the paper with no thought that it would be used to express Lisp programs in practice.

Lisp がチューリングマシンよりも扱いやすいモデルであることを示すもうひとつの方法は、万能 Lisp 関数を書いてそれが万能チューリングマシンの表記よりも簡潔で分かりやすいことを示すことだった。これが Lisp 関数 *eval* だ…、それは Lisp 式の値を計算する。…*eval* を書くために、私は Lisp 関数を Lisp のデータとして表す方法を考えなければならなかった。その表記はあくまで論文のためのもので、実際に Lisp プログラムがそれで書かれるようになるなんて考えもしなかったよ。

What happened next was that, some time in late 1958, Steve Russell, one of McCarthy's grad students, looked at this definition of *eval* and realized that if he translated it into machine language, the result would be a Lisp interpreter.

次に起こったのはこうだ。1958 年後半に、マッカーシーの大学院生の一人であったスティーブ・ラッセルが *eval* の定義を見ていて、これを機械語に変換すれば Lisp インタプリタができるじゃないか、と思い付いたんだ。

This was a big surprise at the time. Here is what McCarthy said about it later in an interview:

これは当時としては画期的なことだった。マッカーシーは後にインタビューでこのように語っている。

Steve Russell said, look, why don't I program this *eval*..., and I said to him, ho, ho, you're confusing theory with practice, this *eval* is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the *eval* in my paper into [IBM] 704 machine code, fixing bugs, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today....

スティーブ・ラッセルが言った。ほら、*eval* をプログラムしたらどうだろう…、で、私は答えたものさ。はは、君は理論と現実を混同しているよ。この *eval* は人が読むためのもので、計算するためのものじゃない。でも彼はそのアイディアを進めて実際にやってしまった。つまり、彼は私の論文の *eval* を [IBM] 704 の機械語に直して、バグを取り、それを Lisp インタプリタと称したんだ。実際そうだった。それが、Lisp が現在の姿を取った瞬間だった。

Suddenly, in a matter of weeks I think, McCarthy found his theoretical exercise transformed into an actual programming language-- and a more powerful one than he had intended.

突如として、おそらく数週間のうちに、マッカーシーは彼の理論的な実験がプログラミング言語に——それも、おそらく彼が意図したのより強力な言語に——変化したのに気付いたのだ。



Lawrence Livermore National Laboratory (LLNL). The IBM 704 Computer (1956).
IBM 704、ローレンスリバモア研究所、1956 年

So the short explanation of why this 1950s language is not obsolete is that it was not technology but math, and math doesn't get stale. The right thing to compare Lisp to is not 1950s hardware, but, say, the Quicksort algorithm, which was discovered in 1960 and is still the fastest general-purpose sort.

だから、なぜ 1950 年代の言語が時代遅れにならないかという簡単な説明は、それが技術じゃなくて数学だったということだ。数学は色褪せない。Lisp を比較する対象は 1950 年代のハードウェアでなく、例えばクイックソートのアルゴリズムだ。それは 1960 年に発見され、未だに汎用ソートアルゴリズムとして最速だ。

There is one other language still surviving from the 1950s, Fortran, and it represents the opposite approach to language design. Lisp was a piece of theory that unexpectedly got turned into a programming language. Fortran was developed intentionally as a programming language, but what we would now

consider a very low-level one.

1950年代から今まで生き延びている、もうひとつの言語がある。Fortran だ。それは Lisp と正反対の方法で設計されている。Lisp は理論の一片が意図せずにプログラミング言語になったものだ。Fortran はプログラミング言語となるべくして設計されたが、現代の我々の基準からすれば非常に低レベルなものだった。

[Fortran I](#), the language that was developed in 1956, was a very different animal from present-day Fortran. Fortran I was pretty much assembly language with math. In some ways it was less powerful than more recent assembly languages; there were no subroutines, for example, only branches. Present-day Fortran is now arguably closer to Lisp than to Fortran I.

1956年に作られた Fortran I は現在の Fortran とはずいぶん違う生物だった。Fortran I はアセンブリ言語に数学がちょっとくっついたものだった。ある意味、現代のアセンブリ言語よりも弱いと言えるかもしれない。例えばサブルーチンはなく分岐だけがあった。現代の Fortran は、Fortran I よりはかなり Lisp に近い。

Lisp and Fortran were the trunks of two separate evolutionary trees, one rooted in math and one rooted in machine architecture. These two trees have been converging ever since. Lisp started out powerful, and over the next twenty years got fast. So-called mainstream languages started out fast, and over the next forty years gradually got more powerful, until now the most advanced of them are fairly close to Lisp. Close, but they are still missing a few things....

Lisp と Fortran は 2 つの独立した進化系統樹の幹にあたる。一方は数学に根ざし、もう一方はマシンアーキテクチャに根ざしている。これらの 2 つの樹は発生以来、常に近づいてきた。Lisp はパワフルな言語として出発し、以降 20 年の間に速度を獲得した。いわゆる主流の言語は高速なものとして出発し、以降 40 年の間に徐々にパワーを得てきた。いまや、最も進んだ言語は Lisp にかなり近い。近い、が、まだいくつか欠けているものがある…。

What Made Lisp Different

Lisp のユニークさ

When it was first developed, Lisp embodied nine new ideas. Some of these we now take for granted, others are only seen in more advanced languages, and two are still unique to Lisp. The nine ideas are, in order of their adoption by the mainstream,

最初に開発された時、Lisp は 9 つの新しいアイディアを内包していた。そのうちのいくつ

かは現代では当然のことになっている。残りのうちいくつかは進んだ言語に見られ、2つはまだ Lisp 特有だ。主流の言語に取り込まれた順にこの 9 つのアイデアを並べるとこうなる。

1. Conditionals. A conditional is an if-then-else construct. We take these for granted now, but Fortran I didn't have them. It had only a conditional goto closely based on the underlying machine instruction.
1. 条件式。if-then-else という構造だ。今ではあって当然のものだが、Fortran I にはなかった。Fortran I が持っていたのは機械語の命令を反映した、条件付き goto であった。
2. A function type. In Lisp, functions are a data type just like integers or strings. They have a literal representation, can be stored in variables, can be passed as arguments, and so on.
2. 関数型。Lisp では関数は整数や文字列と同じようなデータ型のひとつだ。リテラル表記を持ち、変数に代入できて、引数として渡したり、いろいろなことができる。
3. Recursion. Lisp was the first programming language to support it.
3. 再帰。Lisp は再帰をサポートした最初のプログラミング言語だった。
4. Dynamic typing. In Lisp, all variables are effectively pointers. Values are what have types, not variables, and assigning or binding variables means copying pointers, not what they point to.
4. 動的型付け。Lisp では全ての変数は実質的にポインタだ。変数ではなく値のほうに型が付いており、代入や束縛は実質的にポインタのコピーで行われ、ポインタが指されるものには作用しない。
5. Garbage-collection.
5. ガベージコレクション。
6. Programs composed of expressions. Lisp programs are trees of expressions, each of which returns a value. This is in contrast to Fortran and most succeeding languages, which distinguish between expressions and statements.
6. 式でプログラムが構成されること。Lisp プログラムは式の木であり、それぞれの式が値を返す。これは Fortran とそれを引き継ぐ多くの言語とは対照的だ。それらの言語は式と文を区別している。

It was natural to have this distinction in Fortran I because you could not nest statements. And so while you needed expressions for math to work, there was no point in making anything else return a value, because there could not be anything waiting for it.

Fortran Iにおいて式と文を区別することは自然だった。文はネストできなかったからだ。数式が使えるようにするのに式は必要だったが、それ以外の処理が値を返す必要もなかった。文の戻り値を受けるものなんてなかったからだ。

This limitation went away with the arrival of block-structured languages, but by then it was too late. The distinction between expressions and statements was entrenched. It spread from Fortran into Algol and then to both their descendants.

この制限はブロック構造化言語の到来によってなくなるはずだったが、その時には既に手遅れだったのだ。式と文の区別は慣行の中に根を下ろしてしまっていた。それは Fortran から Algol に広がり、その子孫の言語へと受け継がれていった。

7. A symbol type. Symbols are effectively pointers to strings stored in a hash table. So you can test equality by comparing a pointer, instead of comparing each character.
7. シンボル型。シンボルは実質的にはハッシュテーブルに格納された文字列へのポインタである。これによって、文字を一つひとつ比較しないでもポインタの比較だけで済む。
8. A notation for code using trees of symbols and constants.
8. シンボルと定数の木によってコードを表現すること。
9. The whole language there all the time. There is no real distinction between read-time, compile-time, and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime.
9. 言語の全てが常に在ること。読み込み時、コンパイル時、実行時が明確に分離したステップになっていない。読み込み時にコンパイルしてコードを走らせることもできるし、コンパイル時にコードを読んだり走らせたりすることもできるし、実行時にコードを読んだりコンパイルしたりすることができる。

Running code at read-time lets users reprogram Lisp's syntax; running code at compile-time is the basis of macros; compiling at runtime is the basis of Lisp's use as an extension language in programs like Emacs; and reading at runtime

enables programs to communicate using s-expressions, an idea recently reinvented as XML.

読み込み時にコードを走らせることにより、ユーザが Lisp の構文を変更することができる。コンパイル時にコードを走らせるというのはマクロの基本だ。実行時にコンパイルするというのは Lisp を Emacs のようなプログラムの拡張言語として使うことを容易にする。そして、実行時に読み込みができるというのは、プログラム同士が S 式を使って通信できるということだ。最後のアイディアは最近 XML として再発明された。

When Lisp first appeared, these ideas were far removed from ordinary programming practice, which was dictated largely by the hardware available in the late 1950s. Over time, the default language, embodied in a succession of popular languages, has gradually evolved toward Lisp. Ideas 1-5 are now widespread. Number 6 is starting to appear in the mainstream. Python has a form of 7, though there doesn't seem to be any syntax for it.

初めて Lisp が姿を現したとき、これらのアイディアは当時のプログラミング技術から見てあまりにかけ離れていた。1950 年代後半のプログラミングは、その自体のハードウェアによって大きく制限されていたからだ。時が経つにつれ、普通に使われる言語、つまりその時その時で一番の人気を得る言語は、Lisp へと次第に進化してきた。1~5 までのアイディアはいまや広く認知されている。6 番は主流の言語に現れ始めている。Python は 7 番に相当するものを持っているが、そのための構文はないようだ。

As for number 8, this may be the most interesting of the lot. Ideas 8 and 9 only became part of Lisp by accident, because Steve Russell implemented something McCarthy had never intended to be implemented. And yet these ideas turn out to be responsible for both Lisp's strange appearance and its most distinctive features. Lisp looks strange not so much because it has a strange syntax as because it has no syntax; you express programs directly in the parse trees that get built behind the scenes when other languages are parsed, and these trees are made of lists, which are Lisp data structures.

8 番はこの中で最も興味深いものかもしれない。8 番と 9 番のアイディアは、マッカーシーが実装するなんて考えもしなかったことをスティーブ・ラッセルがやってのけたために、全く意図せずに Lisp に入ったものだ。それが結局、Lisp の奇妙な外見と、それを最もユニークたらしめている理由になった。Lisp が奇妙に見えるのは、奇妙な構文を持っているからというより、構文を持っていないからと言ったほうがよいだろう。普通の言語なら舞台裏で構文解析して作られるような構文木を、Lisp プログラマは直接書いてプログラムを表現する。その構文木はリストで表現され、それは Lisp のデータ構造だ。

Expressing the language in its own data structures turns out to be a very powerful feature. Ideas 8 and 9 together mean that you can write programs that write programs. That may sound like a bizarre idea, but it's an everyday thing in Lisp. The most common way to do it is with something called a *macro*.

言語を自分自身のデータ構造で表現するという事は、非常に強力な機能となった。アイデア 8 と 9 を合わせると、プログラムを書くプログラムを書けるということになる。これは突飛なアイデアに聞こえるかもしれないが、Lisp プログラマが日常的に行っていることだ。このアイデアは通常、マクロと呼ばれる方法で実現される。

The term "macro" does not mean in Lisp what it means in other languages. A Lisp macro can be anything from an abbreviation to a compiler for a new language. If you want to really understand Lisp, or just expand your programming horizons, I would [learn more](#) about macros.

Lisp における「マクロ」という用語は他の言語におけるそれとちょっと違う。Lisp のマクロは、単なる省略形から新しい言語のコンパイラまで何にでも成り得る。もし Lisp を本当に理解したいか、プログラミングの認識の地平線を広げたいなら、マクロについて学ぶことをお勧めする。

Macros (in the Lisp sense) are still, as far as I know, unique to Lisp. This is partly because in order to have macros you probably have to make your language look as strange as Lisp. It may also be because if you do add that final increment of power, you can no longer claim to have invented a new language, but only a new dialect of Lisp.

(Lisp の意味での)マクロは、私の知る限り依然として Lisp に特有のものだ。たぶん、マクロを持つためには言語を Lisp と同じような奇妙な外見にしないと駄目だからだろう。それにまた、マクロという最後の力を加えたら、それは新しい言語ではなく Lisp の新しい方言になってしまうからだろう。

I mention this mostly as a joke, but it is quite true. If you define a language that has `car`, `cdr`, `cons`, `quote`, `cond`, `atom`, `eq`, and a notation for functions expressed as lists, then you can build all the rest of Lisp out of it. That is in fact the defining quality of Lisp: it was in order to make this so that McCarthy gave Lisp the shape it has.

私はよくジョークでそういうことを言うのだが、実際それは真実なんだ。`car`、`cdr`、`cons`、`quote`、`cond`、`atom`、`eq`、そして関数をリストで表現する方法を備えた言語があれば、

あなたは Lisp の残り全てをそれで構築することができる。これが Lisp の本質を定義しているんだ。マッカーシーが Lisp にこの姿を与えたのは、まさにそのためだったからだ。

Where Languages Matter

言語が問題になる時

So suppose Lisp does represent a kind of limit that mainstream languages are approaching asymptotically-- does that mean you should actually use it to write software? How much do you lose by using a less powerful language? Isn't it wiser, sometimes, not to be at the very edge of innovation? And isn't popularity to some extent its own justification? Isn't the pointy-haired boss right, for example, to want to use a language for which he can easily hire programmers?

では Lisp が、主流の言語が漸近する極限にあると仮定してみよう。そうだとすると、その言語を使うべきなのだろうか。より力の弱い言語を使ったら、失うものとは何だろうか。場合によっては、最先端にいることを避ける方が賢明なこともあるんじゃないか。それに、人気があることはそれ自体で正当化されないか。例えば髪のとんがった上司がプログラマを簡単に見つけられるような言語を使いたがるのは、それはそれで正しいことなんじゃないか。

There are, of course, projects where the choice of programming language doesn't matter much. As a rule, the more demanding the application, the more leverage you get from using a powerful language. But plenty of projects are not demanding at all. Most programming probably consists of writing little glue programs, and for little glue programs you can use any language that you're already familiar with and that has good libraries for whatever you need to do. If you just need to feed data from one Windows app to another, sure, use Visual Basic.

もちろんプログラミング言語の選択がそんなに問題にならないプロジェクトというのはある。基本的に、アプリケーションの要求が高ければ高いほど、力のある言語を使って得られるものは大きくなる。だがこの世にある多くのプロジェクトの要求はさして高くない。おそらくプログラミングのほとんどは、既存の部品をつなぎ合わせる小さな糊付けプログラムを書くようなものだ。それならもう手に馴染んでいて、目的に合わせたライブラリが充実している言語を使えばいい。Windows のひとつのアプリケーションから別のアプリケーションヘデータを流したいだけなら、そりゃあ Visual Basic を使えばいいのさ。

You can write little glue programs in Lisp too (I use it as a desktop calculator), but the biggest win for languages like Lisp is at the other end of the spectrum, where

you need to write sophisticated programs to solve hard problems in the face of fierce competition. A good example is the [airline fare search program](#) that ITA Software licenses to Orbitz. These guys entered a market already dominated by two big, entrenched competitors, Travelocity and Expedia, and seem to have just humiliated them technologically.

Lisp でも小さな糊付けプログラムを書くことはできる(私はデスクトップで電卓の代わりに使っている)。でも Lisp みたいな言語が一番威力を発揮するのはプログラミングのスペクトルの逆の端、激しい競争に直面しつつ難しい問題を解く洗練されたプログラムを書かなくちゃならない時だ。ITA Software が Orbitz にライセンスした航空料金検索プログラムが良い例だ。彼らは、既に 2 つの大きな競争相手、Travelocity と Expedia がシェアを確立した業界に参入して、技術的に彼らの鼻を明かしたのだ。

The core of ITA's application is a 200,000 line Common Lisp program that searches many orders of magnitude more possibilities than their competitors, who apparently are still using mainframe-era programming techniques. (Though ITA is also in a sense using a mainframe-era programming language.) I have never seen any of ITA's code, but according to one of their top hackers they use a lot of macros, and I am not surprised to hear it.

ITA のアプリケーションのコアは 20 万行の Common Lisp プログラムで、おそらくメインフレーム時代のプログラミングテクニックを使っている競争相手を何桁も上回る組み合わせの中から検索を行う(ITA もある意味ではメインフレーム時代のプログラミング言語を使っていると言えるのだが)。私は ITA のコードを見たことはないが、かの会社のトップハッカーの言によれば、彼らはたくさんのマクロを使っているそうだ。そうだろうと思う。

Centripetal Forces

求心力

I'm not saying there is no cost to using uncommon technologies. The pointy-haired boss is not completely mistaken to worry about this. But because he doesn't understand the risks, he tends to magnify them.

広く使われていない技術を使うのにデメリットがないというつもりはない。かの髪の毛のトンがった上司も、その点を心配するのには一理ある。ただそのリスクを正しく理解しない限り、彼はリスクを過大評価しがちだ。

I can think of three problems that could arise from using less common languages. Your programs might not work well with programs written in other languages. You

might have fewer libraries at your disposal. And you might have trouble hiring programmers.

私思うに、一般的でない言語を使うことから生ずる問題は 3 つある。そのプログラムは別の言語で書かれたプログラムと一緒にうまく動かないかもしれない。すぐ使えるライブラリが少ないかもしれない。そして、プログラマを見つけるのが難しいかもしれない。

How much of a problem is each of these? The importance of the first varies depending on whether you have control over the whole system. If you're writing software that has to run on a remote user's machine on top of a buggy, closed operating system (I mention no names), there may be advantages to writing your application in the same language as the OS. But if you control the whole system and have the source code of all the parts, as ITA presumably does, you can use whatever languages you want. If any incompatibility arises, you can fix it yourself. それぞれはどのくらい問題となるだろうか。最初の問題の重要性は、あなたが全体のシステムをコントロールできるかどうかによって異なってくる。あなたのソフトウェアが、離れたユーザのマシンで、クローズドでバグの多い OS(あえて名前は挙げないが)の上で走らねばならないのだとしたら、OS が書かれているのと同じ言語でアプリケーションを書くことに利点はあるだろう。だが全てのシステムがあなたの制御下にあり、全ての部分のソースコードを持っているなら(おそらく ITA はそうだ)、あなたは望みの言語を使うことができる。互換性に問題が生じたら自分で直せるからだ。

In server-based applications you can get away with using the most advanced technologies, and I think this is the main cause of what Jonathan Erickson calls the "programming language renaissance." This is why we even hear about new languages like Perl and Python. We're not hearing about these languages because people are using them to write Windows apps, but because people are using them on servers. And as software shifts [off the desktop](#) and onto servers (a future even Microsoft seems resigned to), there will be less and less pressure to use middle-of-the-road technologies.

サーバベースのアプリケーションではあなたは最新の技術を使うことができる。これが、ジョナサン・エリクソンが「プログラミング言語のルネッサンス」と言った主要な理由だと思う。Perl や Python といった新しい言語がポピュラーになったのもまさにそのためだ。これらの言語は、人々がそれで Windows アプリケーションを書いているから評判になってるんじゃない。サーバでそれらが使われているから評判になっているんだ。そしてソフトウェアがデスクトップから離れてサーバに移行してゆくにしたがって(マイクロソフトでさえその流れに乗ろうとしている)、主流の技術を使わねばならないというプレッシャーはどん

どん減っていく。

As for libraries, their importance also depends on the application. For less demanding problems, the availability of libraries can outweigh the intrinsic power of the language. Where is the breakeven point? Hard to say exactly, but wherever it is, it is short of anything you'd be likely to call an application. If a company considers itself to be in the software business, and they're writing an application that will be one of their products, then it will probably involve several hackers and take at least six months to write. In a project of that size, powerful languages probably start to outweigh the convenience of pre-existing libraries.

ライブラリに関しても、その重要性は書くアプリケーションに依存する。さほど難しい問題に対しては、ライブラリがあることは言語に備わった本質的な力よりも重要なことだろう。どこで両者の重要性が逆転するだろう?正確に言うのは難しいが、それがどの点であろうとも、アプリケーションと呼べるものに達するちょっと手前であると思う。もしある会社がソフトウェアビジネスで生きていこうと考え、その製品としてアプリケーションを書き始めたとしたら、少なくとも数人のハッカーで半年かそこいらの期間は必要とするはずだ。そのくらいの大きさになれば、既存のライブラリがあることの便利さよりも、言語の力の方が重要になってくるだろう。

The third worry of the pointy-haired boss, the difficulty of hiring programmers, I think is a red herring. How many hackers do you need to hire, after all? Surely by now we all know that software is best developed by teams of less than ten people. And you shouldn't have trouble hiring hackers on that scale for any language anyone has ever heard of. If you can't find ten Lisp hackers, then your company is probably based in the wrong city for developing software.

髪の毛がった上司の 3 番目の懸念、プログラマを見つける難しさに関しては、杞憂だと思う。結局のところ、何人のハッカーが必要なんだい?もう今となっては、ソフトウェアは 10 人以下のチームで書くのが一番だってみんな知っているだろう。そしてどんな言語であっても、名前を聞いたことがあるような言語なら、そのくらいの人数のハッカーは集められるはずだ。10 人の Lisp ハッカーが集められないようなら、たぶんあなたの会社はソフトウェアを開発するのに適さない都市にあるんだ。

In fact, choosing a more powerful language probably decreases the size of the team you need, because (a) if you use a more powerful language you probably won't need as many hackers, and (b) hackers who work in more advanced languages are likely to be smarter.

実際、より力のある言語を選べば、必要なチームの人数を減らすことができるだろう。(a) 言語がパワフルであればそんなに多くのハッカーは必要ないだろうし、(b)より進んだ言語を操るハッカーは一般に能力が高いからだ。

I'm not saying that you won't get a lot of pressure to use what are perceived as "standard" technologies. At Viaweb (now Yahoo Store), we raised some eyebrows among VCs and potential acquirers by using Lisp. But we also raised eyebrows by using generic Intel boxes as servers instead of "industrial strength" servers like Suns, for using a then-obscure open-source Unix variant called FreeBSD instead of a real commercial OS like Windows NT, for ignoring a supposed e-commerce standard called SET that no one now even remembers, and so on.

「標準」と見なされている技術を使うべきだというプレッシャーを全く受けないかと言えばそれは嘘になる。(現在は Yahoo Store となった)Viaweb では、Lisp を使っていることでベンチャーキャピタルや会社を買いに来た人々の目を丸くしたものだ。でも、「実務に耐える」Sun のようなサーバでなく普通の Intel プロセッサ搭載マシンをサーバに使っていたことも、Windows NT のような本物の商用 OS でなく当時はまだあまり知られていなかったオープンソースの Unix である FreeBSD を使っていたことでも、また e-commerce の標準になると思われていた SET(誰も覚えちゃいないと思うが)を無視したことでも、ベンチャーキャピタルの目を丸くしたのだ。

You can't let the suits make technical decisions for you. Did it alarm some potential acquirers that we used Lisp? Some, slightly, but if we hadn't used Lisp, we wouldn't have been able to write the software that made them want to buy us. What seemed like an anomaly to them was in fact cause and effect.

スーツを着た連中に技術に関する決断をさせちゃ駄目だ。私たちが Lisp を使っていることで会社の買収候補者は警戒しなかったかって?そりゃ多少はね。でももし私たちが Lisp を使わなかったとしたら、そもそも彼らを買いたいと思うようなソフトウェアを書くことさえできていなかったんだから。彼らが異常だと思うことは、実は単なる原因と結果に過ぎない。

If you start a startup, don't design your product to please VCs or potential acquirers. *Design your product to please the users.* If you win the users, everything else will follow. And if you don't, no one will care how comfortably orthodox your technology choices were.

ベンチャーを立ち上げるなら、ベンチャーキャピタルや会社の買収候補者を喜ばせようと思って製品をデザインしちゃ駄目だ。ユーザを喜ばせるようにデザインしなきゃ。ユーザ

を獲得すれば、全ては後からついてくる。ユーザがいなけりゃ、技術の選択がどれだけ主流に忠実かなんて誰も気にしやしないさ。

The Cost of Being Average

普通であることのコスト

How much do you lose by using a less powerful language? There is actually some data out there about that.

じゃあ、力の弱い言語を使うことによってどれだけ損をするんだろう。これに関しては多少データがある。

The most convenient measure of power is probably [code size](#). The point of high-level languages is to give you bigger abstractions-- bigger bricks, as it were, so you don't need as many to build a wall of a given size. So the more powerful the language, the shorter the program (not simply in characters, of course, but in distinct elements).

力を測る一番簡単な方法はたぶんコードサイズだろう。高レベル言語を使う理由は、より大きな抽象化を可能にしてくれることだ。より大きな煉瓦を使えば必要な壁を作るのに少ない煉瓦で済む、というようなものだ。だからよりパワフルな言語を使えばプログラムは短くなる(もちろん文字数だけでなく、言語の要素に関してもだ)。

How does a more powerful language enable you to write shorter programs? One technique you can use, if the language will let you, is something called [bottom-up programming](#). Instead of simply writing your application in the base language, you build on top of the base language a language for writing programs like yours, then write your program in it. The combined code can be much shorter than if you had written your whole program in the base language-- indeed, this is how most compression algorithms work. A bottom-up program should be easier to modify as well, because in many cases the language layer won't have to change at all.

よりパワフルな言語を使って、どうやって短いプログラムが書けるんだろう。一つのテクニックとして、もし言語がそれを許すなら、ボトムアッププログラミングと呼ばれるものがある。アプリケーションをベース言語で書く代わりに、まず自分の書きたいプログラムに適合する言語をベース言語で書いて、次にその言語を使ってプログラムを書くんだ。両者を合わせたコードサイズは全てをベース言語で書いた場合よりずっと小さくなるはずだ。実際、これは圧縮アルゴリズムの動作原理と同じだ。ボトムアッププログラムは変更するのも楽だ。多くの変更は言語の層に手を加えなくても済ませられる。

Code size is important, because the time it takes to write a program depends mostly on its length. If your program would be three times as long in another language, it will take three times as long to write-- and you can't get around this by hiring more people, because beyond a certain size new hires are actually a net lose. Fred Brooks described this phenomenon in his famous book *The Mythical Man-Month*, and everything I've seen has tended to confirm what he said.

コードサイズは重要だ。プログラムを書く時間はプログラムの長さに大きく依存する。プログラムが別の言語を使ったら 3 倍のサイズになるんだとしたら、それを書くのに 3 倍の時間が必要になるだろう。そして、それはプログラマを増やすことではカバーできない。あるサイズを越えたら、新しく人員を投入するのは全体として効率を下げるだけになる。フレデリック・ブルックスがこの現象を有名な『人月の神話』で述べているが、私が見聞きした全てのことは彼が言うことを裏付けるものだった。

So how much shorter are your programs if you write them in Lisp? Most of the numbers I've heard for Lisp versus C, for example, have been around 7-10x. But a recent article about ITA in *New Architect* magazine said that "one line of Lisp can replace 20 lines of C," and since this article was full of quotes from ITA's president, I assume they got this number from ITA. If so then we can put some faith in it; ITA's software includes a lot of C and C++ as well as Lisp, so they are speaking from experience.

じゃあ、Lisp で書いたらどのくらいプログラムが短くなるだろう。私が聞いたことがあるのはほとんど Lisp と C の比較だが、例えば、だいたい 7~10 倍という数字だ。だが、最近の『New Architect』誌に掲載された ITA の記事には「一行の Lisp は 20 行の C コードを置き換えられる」とある。この記事は ITA の社長の談話を多く載せているから、たぶんこの数字も ITA から出てきたものだろう。だとしたらある程度信用できる。ITA のソフトウェアは Lisp だけでなく C や C++ のものも多いから、彼らは経験に基づいて語っているのだ。

My guess is that these multiples aren't even constant. I think they increase when you face harder problems and also when you have smarter programmers. A really good hacker can squeeze more out of better tools.

この倍率は定数ではないかもしれない。難しい問題になればなるほど、そしてプログラマが優れていればいるほど、倍率は増加するだろう。良いハッカーであればあるほど、良いツールをもっと活用することができる。

As one data point on the curve, at any rate, if you were to compete with ITA and chose to write your software in C, they would be able to develop software twenty times faster than you. If you spent a year on a new feature, they'd be able to duplicate it in less than three weeks. Whereas if they spent just three months developing something new, it would be *five years* before you had it too.

その曲線の一点として、もしあなたが ITA とあるソフトウェアで競争するとして、あなたが C を選んだら、彼らはあなたの 20 倍速くソフトウェアを開発できるということだ。新しい機能を入れるのにあなたが 1 年を費やしたら、彼らはそれを 3 週間で追加するだろう。彼らが 3 ヶ月かかって新しいものを開発したとしたら、あなたは 5 年かけなければそれに追いつけない。

And you know what? That's the best-case scenario. When you talk about code-size ratios, you're implicitly assuming that you can actually write the program in the weaker language. But in fact there are limits on what programmers can do. If you're trying to solve a hard problem with a language that's too low-level, you reach a point where there is just too much to keep in your head at once.

そしてだよ。それはベストケースの想定だ。こうやってコードサイズの比較をしている時は、一応弱い言語でもソフトは書き上げられると仮定している。でも現実には、プログラマができることには限界があるんだ。低レベルすぎる言語で難しい問題を解こうとしたら、ある時点で頭の中に納まりきらなくなる。

So when I say it would take ITA's imaginary competitor five years to duplicate something ITA could write in Lisp in three months, I mean five years if nothing goes wrong. In fact, the way things work in most companies, any development project that would take five years is likely never to get finished at all.

先ほどの ITA の仮想の競争相手が、ITA が Lisp で 3 ヶ月で書いたものを 5 年かければコピーできると言ったのは、あくまで全てがうまくいった場合の話だ。実際には、多くの会社では、5 年もかかるようなプロジェクトはまず完成しない。

I admit this is an extreme case. ITA's hackers seem to be unusually smart, and C is a pretty low-level language. But in a competitive market, even a differential of two or three to one would be enough to guarantee that you'd always be behind. もちろんこれは極端な例だ。ITA のハッカーたちはものすごく優秀だし、C はあまりに低レベルな言語だ。だが競争の激しい市場では、2~3 対 1 という差であっても、ライバルに置いていかれるには十分な数字だ。

A Recipe

レシピ

This is the kind of possibility that the pointy-haired boss doesn't even want to think about. And so most of them don't. Because, you know, when it comes down to it, the pointy-haired boss doesn't mind if his company gets their ass kicked, so long as no one can prove it's his fault. The safest plan for him personally is to stick close to the center of the herd.

髪のとんがった上司にとって、こんなことは考えたくもないことだろう。だからたいていの上司は考えない。結局のところ、彼らは会社が多少どうなろうと自分の責任にならなければよいのだ。彼にとって安全なプランとは、なるべく群の中心に近い位置にいることだ。

Within large organizations, the phrase used to describe this approach is "industry best practice." Its purpose is to shield the pointy-haired boss from responsibility: if he chooses something that is "industry best practice," and the company loses, he can't be blamed. He didn't choose, the industry did.

大きな組織では、このアプローチは「業界のベストプラクティス」という言葉で表される。その目的は髪のとんがった上司を責任から守ることだ。彼が「業界のベストプラクティス」である何かを選んでそれで会社が競争に負けたら、悪いのは彼じゃない。彼が選んだのではなく、業界が選んだのだから。

I believe this term was originally used to describe accounting methods and so on. What it means, roughly, is *don't do anything weird*. And in accounting that's probably a good idea. The terms "cutting-edge" and "accounting" do not sound good together. But when you import this criterion into decisions about technology, you start to get the wrong answers.

この言葉は、たぶんもともとは会計方法かなんかから出てきたんじゃないかと思う。その言葉の意味を乱暴に言えば、「妙なことはするな」ということだ。会計の世界ではたぶんそれは良いアイディアだ。でも「最先端」と「会計」はあんまり馴染まない。だからそんな方法を技術に関する決断に持ち込んだら、間違った答えを得るようになる。

Technology often *should* be cutting-edge. In programming languages, as Erann Gat has pointed out, what "industry best practice" actually gets you is not the best, but merely the average. When a decision causes you to develop software at a fraction of the rate of more aggressive competitors, "best practice" is a misnomer.

技術はしばしば、最先端でなければならない。プログラミング言語では、エラン・ガットが示したように、「業界のベストプラクティス」はあなたをトップにするのではなく、単に平凡にするだけだ。その決断のせいでより進んだ競争相手の数分の一しか開発ができないのなら、「ベストプラクティス」は誤った名前だとさえ言える。

So here we have two pieces of information that I think are very valuable. In fact, I know it from my own experience. Number 1, languages vary in power. Number 2, most managers deliberately ignore this. Between them, these two facts are literally a recipe for making money. ITA is an example of this recipe in action. If you want to win in a software business, just take on the hardest problem you can find, use the most powerful language you can get, and wait for your competitors' pointy-haired bosses to revert to the mean.

ここで、私は 2 つの情報を述べた。どちらもとても貴重な情報だと思う。私はそれを自分の経験から得た。第一に、プログラミング言語はその力に差がある。第二に、たいていのマネージャはその事実を無視する。実はこの 2 つは、文字通りお金を儲けるレシピになるんだ。ITA は、このレシピがまさに使われている良い例だ。ソフトウェアビジネスで勝とうと思ったら、見つけられるうちで一番難しい問題に目を付けて、手に入る一番パワフルな言語を使い、そしてライバル会社の髪のとんがった上司が中庸な技術へと戻って行くのを待っていればいいのさ。

Appendix: Power

付録：力

As an illustration of what I mean about the relative power of programming languages, consider the following problem. We want to write a function that generates accumulators-- a function that takes a number n , and returns a function that takes another number i and returns n incremented by i .

私が言う言語の相対的な力を説明するために、次の問題を考えてみよう。アキュムレータを生成する関数、すなわち、数 n を取り、「数 i を取って n を i だけ増加させ、その増加した値を返す関数」を返すような関数だ。

(That's *incremented by*, not plus. An accumulator has to accumulate.)

(「増加させる」に注意。ただ足すだけではない。アキュムレータ(累積器)だから累積させなければ。)

In Common Lisp this would be
Common Lisp ではこんなふうになるだろう。

```
(defun foo (n)
  (lambda (i) (incf n i)))
```

and in Perl 5,
一方、Perl 5 ではこうだ。

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```

which has more elements than the Lisp version because you have to extract parameters manually in Perl.

Perl 5 のほうが Lisp より要素数が多いが、これは Perl ではパラメータを自分で取り出さなければならぬからだ。

In Smalltalk the code is slightly longer than in Lisp
Smalltalk も Lisp よりちょっと長くなる。

```
foo: n
  |s|
  s := n.
  ^[:i| s := s+i. ]
```

because although in general lexical variables work, you can't do an assignment to a parameter, so you have to create a new variable s.

レキシカルな変数を使えるが、パラメータには代入できないから、新しい変数 s を作らなくちゃならない。

In Javascript the example is, again, slightly longer, because Javascript retains the distinction between statements and expressions, so you need explicit return statements to return values:

JavaScript でもこの例はちょっと長くなるが、それは文と式の区別を持っているために、明示的な `return` 文を使わなければならないからだ。

```
function foo(n) {  
  return function (i) {  
    return n += i } }  
}
```

(To be fair, Perl also retains this distinction, but deals with it in typical Perl fashion by letting you omit returns.)

(公正を期するなら、Perl もこの区別を持っているが、通常書き方では `return` を省略できる。)

If you try to translate the Lisp/Perl/Smalltalk/Javascript code into Python you run into some limitations. Because Python doesn't fully support lexical variables, you have to create a data structure to hold the value of `n`. And although Python does have a function data type, there is no literal representation for one (unless the body is only a single expression) so you need to create a named function to return. This is what you end up with:

この Lisp/Perl/Smalltalk/JavaScript を Python に移そうとするとちょっとした制限につきあたる。Python には完全なレキシカル変数がないため、`n` の値を保持するデータ構造を作らなければならないからだ。また、Python は関数データ型を持っているが、その関数が単一の式で表されない場合はリテラル表記できない。だから名前を付けた関数を作ってそれを返さなければならない。こんな感じになるだろう。

```
def foo(n):  
    s = [n]  
    def bar(i):  
        s[0] += i  
        return s[0]  
    return bar
```

Python users might legitimately ask why they can't just write
Python ユーザはなぜ以下のように書けないか疑問に思っただろう。

```
def foo(n):  
    return lambda i: return n += i
```

or even

あるいは、

```
def foo(n):  
    lambda i: n += i
```

and my guess is that they probably will, one day. (But if they don't want to wait for Python to evolve the rest of the way into Lisp, they could always just...)

そして、たぶんいつの日か、そう書けるようになるだろう。(ただ、Python が Lisp への最後の道程を進化するのを待てないユーザには、いつでも使える手段がある。それは…)

In OO languages, you can, to a limited extent, simulate a closure (a function that refers to variables defined in enclosing scopes) by defining a class with one method and a field to replace each variable from an enclosing scope. This makes the programmer do the kind of code analysis that would be done by the compiler in a language with full support for lexical scope, and it won't work if more than one function refers to the same variable, but it is enough in simple cases like this.

オブジェクト指向言語では、ひとつだけメソッドを持つクラスを定義してメソッドが参照する変数をクラスのフィールドに置き換えれば、制限はあるものの、クロージャ(定義されたスコープ内の変数を参照できる関数)を模倣することはできる。そうすることは、レキシカルスコープを完全にサポートする言語ではコンパイラがやってくれるような作業をプログラマが自分でやるようなものだし、複数の関数が同一の変数を参照する場合なんかは使えないけれど、この例のような簡単な場合は十分だ。

Python experts seem to agree that this is the preferred way to solve the problem in Python, writing either

Python のエキスパートは、こっちの方法のほうが Python らしいということに同意するんじゃないだろうか。例えばこう書いたり

```
def foo(n):  
    class acc:  
        def __init__(self, s):  
            self.s = s  
        def inc(self, i):  
            self.s += i  
            return self.s
```

```
return acc(n).inc
```

or

あるいはこうだ。

```
class foo:
    def __init__(self, n):
        self.n = n
    def __call__(self, i):
        self.n += i
        return self.n
```

I include these because I wouldn't want Python advocates to say I was misrepresenting the language, but both seem to me more complex than the first version. You're doing the same thing, setting up a separate place to hold the accumulator; it's just a field in an object instead of the head of a list. And the use of these special, reserved field names, especially `__call__`, seems a bit of a hack. 私がこの例を入れたのは、Python 支持者が、私が言語を間違った方法で紹介しているんだと思わないようにだ。だがどちらの方法も、私にとっては最初の例より複雑に見える。アキュムレータを保持する場所を作るという同じことをしているだけで、それが最初の例ではリストの先頭であり、これらの例ではフィールドになっただけだ。それに特殊な予約フィールド名、特に `__call__` はちょっとしたハックのように感じられる。

In the rivalry between Perl and Python, the claim of the Python hackers seems to be that that Python is a more elegant alternative to Perl, but what this case shows is that power is the ultimate elegance: the Perl program is simpler (has fewer elements), even if the syntax is a bit uglier.

Perl と Python の比較では、Python ハッカーのほうはいつも Python が Perl よりエレガントに書けると主張しているようだが、この例を見る限り、言語の力は究極のエレガンスに通じるようだ。Perl のほうが、構文はちょっと醜いが、単純だ(より少ない要素で書ける)。

How about other languages? In the other languages mentioned in this talk-- Fortran, C, C++, Java, and Visual Basic-- it is not clear whether you can actually solve this problem. Ken Anderson says that the following code is about as close as you can get in Java:

他の言語ではどうだろう。この話で述べた他の言語(**Fortran**、**C**、**C++**、**Java**、そして **Visual Basic**)では、この問題が解けるかどうかさえ定かではない。ケン・アンダーソンは、次のコードは **Java** でやる限り最も近い解だと述べた。

```
public interface Inttoint {
    public int call(int i);
}

public static Inttoint foo(final int n) {
    return new Inttoint() {
        int s = n;
        public int call(int i) {
            s = s + i;
            return s;
        }
    };
}
```

This falls short of the spec because it only works for integers. After many email exchanges with Java hackers, I would say that writing a properly polymorphic version that behaves like the preceding examples is somewhere between damned awkward and impossible. If anyone wants to write one I'd be very curious to see it, but I personally have timed out.

これは整数でしか動作しないという点で元のスペックに届いていない。**Java** ハッカーとずいぶんメールで討論したが、上のほうで示したような例と同様に動作する真にポリモーフィックなコードを **Java** で書くのは、ものすごく不自由と不可能との間だ、ということのようだ。誰かうまく書けたらぜひ見せてほしい。私は自分でもやってみたが時間切れだった。

It's not literally true that you can't solve this problem in other languages, of course. The fact that all these languages are Turing-equivalent means that, strictly speaking, you can write any program in any of them. So how would you do it? In the limit case, by writing a Lisp interpreter in the less powerful language.

もちろん、他の言語でこの問題を解けないと言ってしまうのは正しくはない。全ての言語はチューリング等価であるという事実は、厳密に言えば、いかなるプログラムもどんな言語を使っても書けるということだ。じゃあどうやって?このようなケースでは、力の弱い言語で **Lisp** インタプリタを書くことで実現できる。

That sounds like a joke, but it happens so often to varying degrees in large programming projects that there is a name for the phenomenon, Greenspun's Tenth Rule:

これはジョークのように思えるかもしれないが、そういうことは大きなプロジェクトでは程度の差こそあれ非常に頻繁に起こっていることだ。だから名前さえ付いている。「グリーンスパンの第 10 規則」というものだ。

Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

全ての十分に複雑な C もしくは Fortran プログラムは、後付けの、不完全な仕様とバグを持ち、遅い、Common Lisp の半分の実装を含んでいる。

If you try to solve a hard problem, the question is not whether you will use a powerful enough language, but whether you will (a) use a powerful language, (b) write a de facto interpreter for one, or (c) yourself become a human compiler for one. We see this already beginning to happen in the Python example, where we are in effect simulating the code that a compiler would generate to implement a lexical variable.

あなたが難しい問題を解こうとしているなら、問われているのはパワフルな言語を使うか使わないか、ではない。(a)パワフルな言語を使うか、(b)パワフルな言語と等価なインタプリタを書くか、(c)自らがパワフルな言語の人間コンパイラとなるか、という選択なのだ。Python の例の中に既にこの選択を見て取ることができる。あの例で私たちは、コンパイラがレキシカルな変数を扱う際に生成するようなコードをシミュレートしていたのだ。

This practice is not only common, but institutionalized. For example, in the OO world you hear a good deal about "patterns". I wonder if these patterns are not sometimes evidence of case (c), the human compiler, at work. When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough--often that I'm generating by hand the expansions of some macro that I need to write.

コンパイラがやるべきことを人間がシミュレートするという慣行はただ広まっているというだけでなく、思考を型にはめる作用がある。例えば、オブジェクト指向の世界では非常によく「パターン」というのを耳にするだろう。この「パターン」は多くの場合、(c)のケ

ース、すなわち人間コンパイラが実際に動作している証拠なんじゃないかと私は思う。私が自分のプログラムにパターンを見つけたら、それはどこかがおかしいというサインだ。プログラムの形は、それが解くべき問題のみを反映すべきだ。その他の繰り返しがコード中に現れるということは、少なくとも私にとっては、十分な抽象化を行っていないということの意味する。たいていの場合、それはマクロを書くべきコードを手で拡張して書いているということになる。

Notes

注釈

- The IBM 704 CPU was about the size of a refrigerator, but a lot heavier. The CPU weighed 3150 pounds, and the 4K of RAM was in a separate box weighing another 4000 pounds. The Sub-Zero 690, one of the largest household refrigerators, weighs 656 pounds.
- IBM 704 の CPU はだいたい冷蔵庫の大きさだったが、ずっと重かった。CPU それ自体が 3150 ポンド(約 1.4t)の重さだったし、4K の RAM が別の箱に入っていて 4000 ポンドだった。ちなみに家で使われる最大型の冷蔵庫である Sub-Zero 690 は 656 ポンドである。
- Steve Russell also wrote the first (digital) computer game, Spacewar, in 1962.
- スティーブ・ラッセルはまた、1962 年に最初の(デジタル)コンピュータゲームである Spacewar も書いた。
- If you want to trick a pointy-haired boss into letting you write software in Lisp, you could try telling him it's XML.
- もしあなたが、髪の毛のとんがった上司をうまく説得して Lisp でソフトウェアを書かせてもらおうとするなら、それは XML だと言ってみるのも手かもしれない。
- Here is the accumulator generator in other Lisp dialects:
- これが、他の Lisp 方言におけるアキュムレータジェネレータのコードだ。

```
Scheme: (define (foo n)
          (lambda (i) (set! n (+ n i)) n))
Goo:    (df foo (n) (op incf n _)))
Arc:    (def foo (n) [++ n _])
```

- Erann Gat's sad tale about "industry best practice" at JPL inspired me to address this generally misapplied phrase.
- エラン・ガットの JPL における「業界のベストプラクティス」についての悲しい話を聞いて、この言葉の間違った適用を正しておかねばと思ったのだ。
- Peter Norvig found that 16 of the 23 patterns in *Design Patterns* were "[invisible or simpler](#)" in Lisp.
- ピーター・ノーヴィッグは『デザインパターン』に挙げられた 23 のパターンのうち 16 は Lisp では「全く見えないか、あるいはより簡潔である」ことを見出した。
- **Thanks** to the many people who answered my questions about various languages and/or read drafts of this, including Ken Anderson, Trevor Blackwell, Erann Gat, Dan Giffin, Sarah Harlin, Jeremy Hylton, Robert Morris, Peter Norvig, Guy Steele, and Anton van Straaten. They bear no blame for any opinions expressed.

Related

関連情報

Many people have responded to this talk, so I have set up an additional page to deal with the issues they have raised: [Re: Revenge of the Nerds](#).

この話はたくさんの人々から反響があったので、議論に上がったことがらに関して追加でページを設けた「オタク野郎の復讐」への反響。

It also set off an extensive and often useful discussion on the [LL1](#) mailing list. See particularly the mail by Anton van Straaten on semantic compression.

また、LL1 メーリングリストにおいてもしばしば長く有用な議論に火をつけた。特に、アントン・ファン・ストラーテンのセマンティックコンプレッションのメールを参照のこと。

Some of the mail on LL1 led me to try to go deeper into the subject of language power in [Succinctness is Power](#).

LL1 でのいくつかのメールに触発されて、私は言語の力の問題について「簡潔さは力なり」でより深く考えてみた。

A larger set of canonical implementations of the [accumulator generator](#)

[benchmark](#) are collected together on their own page.

アキュムレータジェネレータのベンチマークに関するより多くの正規実装は独自のページにまとめている。

Chapter 14 THE DREAM LANGUAGE (BEING POPULAR)

第 14 章 夢の言語 (人気の言語を作るには)

May 2001

(This article was written as a kind of business plan for a [new language](#). So it is missing (because it takes for granted) the most important feature of a good programming language: very powerful abstractions.)

(この記事は、新しい Lisp のための一種のビジネスプランとしてまとめたものである。したがって、良いプログラミング言語の最も重要な機能である、強力な抽象化機能という点は省いてある(当然備えているべきものだからだ)。)

A friend of mine once told an eminent operating systems expert that he wanted to design a really good programming language. The expert told him that it would be a waste of time, that programming languages don't become popular or unpopular based on their merits, and so no matter how good his language was, no one would use it. At least, that was what had happened to the language *he* had designed.

友人がかつて、著明なオペレーティングシステムの専門家に、本当に素晴らしいプログラミング言語を設計したいんだと話したことがある。専門家は、それは時間の無駄だよと答えた。プログラミング言語に人気が出るかどうかはその言語の良さとは関係がない、だからいかに良い言語を作ろうと誰も使ってはくれないよ、と。少なくとも、それが彼がかつて設計した言語に起こったことだった。

What does make a language popular? Do popular languages deserve their popularity? Is it worth trying to define a good programming language? How would you do it?

何がある言語を人気のあるものにするのだろうか? 人気のある言語はその人気に見合っているのだろうか。良いプログラミング言語というものを定義する試みに意味はあるだろうか。どうやればそれができるだろうか。

I think the answers to these questions can be found by looking at hackers, and learning what they want. Programming languages are *for* hackers, and a programming language is good as a programming language (rather than, say, an exercise in denotational semantics or compiler design) if and only if hackers like it. これらの質問の答えは、ハッカーを観察し、彼らが何を欲しているかを学ぶことで得られると私は考える。プログラミング言語はハッカーのものだ。プログラミング言語は、ハッ

カーに好まれてこそプログラミング言語たることができる。そうでなければ表示意味論やコンパイラ設計の単なる練習課題にすぎない。

1 The Mechanics of Popularity

1 人気の仕組み

It's true, certainly, that most people don't choose programming languages simply based on their merits. Most programmers are told what language to use by someone else. And yet I think the effect of such external factors on the popularity of programming languages is not as great as it's sometimes thought to be. I think a bigger problem is that a hacker's idea of a good programming language is not the same as most language designers'.

確かに、たいていの人々はプログラミング言語を、その利点だけを見て選ぶことはしない。プログラマの多くは他人から使うべき言語を指示される。それでも、そのような外部の要因がプログラミング言語の人気に及ぼす影響は、普通に考えられているほど大きくないと私は思う。むしろ大きな問題は、ハッカーが考える良いプログラミング言語というものと、多くの言語設計者が考えるそれとが同じでないことだ。

Between the two, the hacker's opinion is the one that matters. Programming languages are not theorems. They're tools, designed for people, and they have to be designed to suit human strengths and weaknesses as much as shoes have to be designed for human feet. If a shoe pinches when you put it on, it's a bad shoe, however elegant it may be as a piece of sculpture.

ハッカーと言語設計者では、重要なのはハッカーの意見のほうだ。プログラミング言語は定理ではない。それは道具であり、人々のために設計されるべきだ。靴が人間の足のためにデザインされるのと全く同様に、人間の強さと弱さに合致するようにデザインされなくちゃならない。彫刻作品と見まがうほど優美な外観を備えていようと、履いたら足が痛くなるようなら、それは悪い靴でしかない。

It may be that the majority of programmers can't tell a good language from a bad one. But that's no different with any other tool. It doesn't mean that it's a waste of time to try designing a good language. [Expert hackers](#) can tell a good language when they see one, and they'll use it. Expert hackers are a tiny minority, admittedly, but that tiny minority write all the good software, and their influence is such that the rest of the programmers will tend to use whatever language they use. Often, indeed, it is not merely influence but command: often the expert

hackers are the very people who, as their bosses or faculty advisors, tell the other programmers what language to use.

プログラムの大部分は良い言語と悪い言語の区別がつかないのかもしれない。が、それは他の道具として同じことだ。多くの人に見分けがつかないからといって良い言語を設計しようとするのが時間の無駄になるなんてことはない。優れたハッカーは良い言語を見ればそれと分かるし、それを使おうとするはずだ。優れたハッカーはごく少数派ではあるが、その少数の人々が優れたソフトウェアを生み出し、残りのプログラマがどの言語を使うかに影響を与えていくのだ。実際のところ、それは影響じゃなくて命令になる場合だってある。優れたハッカーは上司や教員として、他のプログラマにどの言語を使うか指示する立場にすることが多いからだ。

The opinion of expert hackers is not the only force that determines the relative popularity of programming languages-- legacy software (Cobol) and hype (Ada, Java) also play a role-- but I think it is the most powerful force over the long term. Given an initial critical mass and enough time, a programming language probably becomes about as popular as it deserves to be. And popularity further separates good languages from bad ones, because feedback from real live users always leads to improvements. Look at how much any popular language has changed during its life. Perl and Fortran are extreme cases, but even Lisp has changed a lot. Lisp 1.5 didn't have macros, for example; these evolved later, after hackers at MIT had spent a couple years using Lisp to write real programs. [\[1\]](#)

優れたハッカーの意見だけがプログラミング言語の相対的な人気を決めるわけではない。過去の遺産(Cobol)や大げさな宣伝(Ada、Java)ももちろん影響する。だが長期的な影響力としては、やはり優れたハッカーの意見が最も強力だと私は考える。臨界点となる初期ユーザ層と十分な時間があれば、プログラミング言語はその力にふさわしい人気を得ることができるはずだ。そして、人気が出れば出るほど良い言語は悪い言語との差を拡げて行く。現実の、生きたユーザからのフィードバックが常に発展を促すからだ。人気のある言語がいかに変化していったかを見たまえ。Perl と Fortran が最も極端なケースだが、Lisp でさえ大きく変化したのだ。例えば、Lisp 1.5 はマクロを持っていなかった。MIT のハッカーたちが本物のプログラムを書くために Lisp を 2 年ほどいじった後でマクロは追加されたのだ。 [\[1\]](#)

So whether or not a language has to be good to be popular, I think a language has to be popular to be good. And it has to stay popular to stay good. The state of the art in programming languages doesn't stand still. And yet the Lisps we have today are still pretty much what they had at MIT in the mid-1980s, because that's the

last time Lisp had a sufficiently large and demanding user base.

したがって、人気が出るためには良い言語でなければならないかどうかはともかくとして、良い言語であるためには人気がなければだめだ。そして、良い言語であり続けるためには人気があり続けなければならない。最先端のプログラミング言語はじっとしていない。こんにちの Lisp は 1980 年代中頃に MIT にあったものからたいして変化していないが、それはその時代を最後に Lisp は発展に必要な大きさのユーザベースを失ったからだ。

Of course, hackers have to know about a language before they can use it. How are they to hear? From other hackers. But there has to be some initial group of hackers using the language for others even to hear about it. I wonder how large this group has to be; how many users make a critical mass? Off the top of my head, I'd say twenty. If a language had twenty separate users, meaning twenty users who decided on their own to use it, I'd consider it to be real.

もちろん、ハッカーは言語を使う前にそれを知らなければならない。どうやって彼らは新しい言語を知るのか。他のハッカーからだ。だがそれには、一番最初にその言語を使い始めるハッカーたちがいなければならない。この最初のハッカーたちのグループはどのくらいの大きさだろう。臨界点となるためには何人くらいのユーザが必要だろう。私の頭にぱっと浮かんだ答えは、20 人だ。もし言語に 20 人の独立したユーザ、つまり自分でその言語を使うと決めたユーザがいれば、私はその言語が本物だと考える。

Getting there can't be easy. I would not be surprised if it is harder to get from zero to twenty than from twenty to a thousand. The best way to get those initial twenty users is probably to use a trojan horse: to give people an application they want, which happens to be written in the new language.

そこに至るのは簡単ではない。0 から 20 に至るのは、おそらく 20 から 1000 に至るのより難しいだろう。最初の 20 人を獲得する最も良い方法はトロイの木馬だ。人々の欲するアプリケーションを、たまたまその言語で書いて提供するのだ。

2 External Factors

2 外部の要因

Let's start by acknowledging one external factor that does affect the popularity of a programming language. To become popular, a programming language has to be the scripting language of a popular system. Fortran and Cobol were the scripting languages of early IBM mainframes. C was the scripting language of Unix, and so, later, was Perl. Tcl is the scripting language of Tk. Java and Javascript are intended

to be the scripting languages of web browsers.

まず、プログラミング言語の人気を大きく左右する外部要因のひとつから話を始めよう。人気が出るためには、プログラミング言語は人気のあるシステムのスクリプト言語でなければならない。Fortran と Cobol は初期の IBM メインフレームのスクリプト言語だった。C は Unix のスクリプト言語だったし、後に Perl がそうになった。Tcl は Tk の、Java と JavaScript は Web ブラウザのスクリプト言語だ。

Lisp is not a massively popular language because it is not the scripting language of a massively popular system. What popularity it retains dates back to the 1960s and 1970s, when it was the scripting language of MIT. A lot of the great programmers of the day were associated with MIT at some point. And in the early 1970s, before C, MIT's dialect of Lisp, called MacLisp, was one of the only programming languages a serious hacker would want to use.

Lisp は残念ながらものすごく人気のあるシステムのスクリプト言語ではなく、したがってものすごく人気のある言語に成り得ていない。現在の Lisp の人気は、Lisp が MIT のスクリプト言語であった 1960 年代と 1970 年代の頃の名残りでしかない。当時の多くの偉大なプログラマはどこかで MIT とつながっている。そして 1970 年代の初期、C の前に、MIT の Lisp 方言である MacLisp は真剣なハッカーが使いたがる唯一の言語であった。

Today Lisp is the scripting language of two moderately popular systems, Emacs and Autocad, and for that reason I suspect that most of the Lisp programming done today is done in Emacs Lisp or AutoLisp.

こんにちでは、Lisp はやや人気のある二つのシステム、Emacs と Autocad のスクリプト言語である。おそらく、こんにちの Lisp プログラミングの大部分は Emacs Lisp と AutoLisp で行われているだろう。

Programming languages don't exist in isolation. To hack is a transitive verb-- hackers are usually hacking something-- and in practice languages are judged relative to whatever they're used to hack. So if you want to design a popular language, you either have to supply more than a language, or you have to design your language to replace the scripting language of some existing system.

プログラミング言語は孤立しては存在しない。「ハックする」は他動詞だ。ハッカーは普通何かをハックする。そして現実にはプログラミング言語はそれを使って何をハックするかによって判断される。だから人気のある言語を設計しようと思ったら、言語以上のものを提供するか、既にあるシステムのスクリプト言語を置き換えるかしなくちゃならない。

Common Lisp is unpopular partly because it's an orphan. It did originally come with a system to hack: the Lisp Machine. But Lisp Machines (along with parallel computers) were steamrolled by the increasing power of general purpose processors in the 1980s. Common Lisp might have remained popular if it had been a good scripting language for Unix. It is, alas, an atrociously bad one.

Common Lisp に人気が出ない理由のひとつは、それが孤児だからだ。Common Lisp は元々はそれでハックするシステムである Lisp Machine から生まれた。しかし、Lisp Machine は(並列計算機とともに)、1980 年代の汎用プロセッサの増大するパワーによってペしゃんこにされてしまった。Common Lisp が Unix の良いスクリプト言語であればまだ人気を保っていたかもしれないが、残念ながら Common Lisp はスクリプト言語としては猛烈にひどい出来だ。

One way to describe this situation is to say that a language isn't judged on its own merits. Another view is that a programming language really isn't a programming language unless it's also the scripting language of something. This only seems unfair if it comes as a surprise. I think it's no more unfair than expecting a programming language to have, say, an implementation. It's just part of what a programming language is.

このことを指して、やっぱり言語はそれ自身の良さだけでは判断されないじゃないかと思うかもしれない。しかし、プログラミング言語は何かのスクリプト言語でなければ真のプログラミング言語になれないんだ、と考えることもできる。このことに今気付いた人にとっては、これはアンフェアに思えるかもしれない。私は、それは、プログラミング言語には実装が伴っていないなければならないというのと同じくらいフェアな意見だと考えている。それはプログラミング言語というものの一部なのだ。

A programming language does need a good implementation, of course, and this must be free. Companies will pay for software, but individual hackers won't, and it's the hackers you need to attract.

もちろん、プログラミング言語は良い実装を必要とするし、しかもそれはフリーでなければならない。会社はソフトウェアに金を払うだろうが、個々のハッカーは払わない。そして今惹き付けたいのはハッカーなのだ。

A language also needs to have a book about it. The book should be thin, well-written, and full of good examples. K&R is the ideal here. At the moment I'd almost say that a language has to have a book published by O'Reilly. That's becoming the test of mattering to hackers.

また、言語についての良い本も必要だ。その本は薄くてうまく書かれていて、また良い例がいっぱい詰まっていなければならない。カーニハンとリッチーの『プログラミング言語 C』は理想的だ。現時点で言うなら、言語にはオライリーから出版された本がなければならないとも言えるだろう。それはハッカーの関心を得るテストになりつつある。

There should be online documentation as well. In fact, the book can start as online documentation. But I don't think that physical books are outmoded yet. Their format is convenient, and the de facto censorship imposed by publishers is a useful if imperfect filter. Bookstores are one of the most important places for learning about new languages.

オンラインドキュメントも必要だ。実際、本は最初はオンラインドキュメントとして書かれ始めたっていい。だが物理的な本の必要性はまだなくなっていないと思う。本という形式は便利だし、完璧ではないにせよ出版社を通ったということである程度の質は保証される。本屋は依然として新しい言語を学ぶのに最も重要な場所だ。

3 Brevity

3 簡潔さ

Given that you can supply the three things any language needs-- a free implementation, a book, and something to hack-- how do you make a language that hackers will like?

どんな言語にも必要な 3 要素、つまり、フリーの実装、本、そしてハックすべきものが揃ったとして、さてどうやって言語をハッカーの好むものにできるだろう。

One thing hackers like is brevity. Hackers are lazy, in the same way that mathematicians and modernist architects are lazy: they hate anything extraneous. It would not be far from the truth to say that a hacker about to write a program decides what language to use, at least subconsciously, based on the total number of characters he'll have to type. If this isn't precisely how hackers think, a language designer would do well to act as if it were.

ハッカーが好む要素のひとつは簡潔さだ。ハッカーは怠惰だ。数学者やモダニズム建築家が怠惰であるのと同じように、余分なものは嫌われる。これからプログラムを書こうとしているハッカーは少なくとも無意識的に予測される総タイプ量によってプログラミング言語を選んでいて、と言っても真実からそれほど遠くはあるまい。これがハッカーの考える方法と正確に一致していないとしても、言語設計者はそう仮定することでうまくやれるだろう。

It is a mistake to try to baby the user with long-winded expressions that are meant to resemble English. Cobol is notorious for this flaw. A hacker would consider being asked to write

英語に似せた長くてごちゃごちゃした表現でユーザを子供扱いするのは間違いだ。Cobol はこの欠陥によって悪名が高い。ハッカーは、

add x to y giving z

instead of

と書く代わりに

$z = x + y$

as something between an insult to his intelligence and a sin against God.

と書けなんて言われたら、良くて彼の知能への侮辱、悪くて神をも恐れぬ大罪だと思うだろう。

It has sometimes been said that Lisp should use first and rest instead of car and cdr, because it would make programs easier to read. Maybe for the first couple hours. But a hacker can learn quickly enough that car means the first element of a list and cdr means the rest. Using first and rest means 50% more typing. And they are also different lengths, meaning that the arguments won't line up when they're called, as car and cdr often are, in successive lines. I've found that it matters a lot how code lines up on the page. I can barely read Lisp code when it is set in a variable-width font, and friends say this is true for other languages too. プログラムを読みやすくするために、Lisp は car と cdr の代わりに first と rest を使うべきだとしばしば言われてきた。まあ、最初の 2 時間くらいはそうかもしれない。だがハッカーは、car はリストの最初の要素で cdr は残りの要素だなんてことくらいすぐに覚えられる。first と rest は 50% もタイプ量が多い。それに first と rest じゃ長さが違うから、並べて使われた時に car と cdr のように引数が綺麗に並ばない。ページ上でどうコードが並ぶかというのは重要だと私は経験から知った。可変幅のフォントで表示された Lisp コードを私はほとんどともに読めないし、私の友人たちはそれは他の言語でも同様だと言う。

Brevity is one place where strongly typed languages lose. All other things being equal, no one wants to begin a program with a bunch of declarations. Anything

that can be implicit, should be.

簡潔さという点では、静的に型付けされた言語は負ける。他の条件が同じなら、誰もたくさんの宣言の並びからプログラムを書き始めたいとは思わない。暗黙にできることは暗黙になっているべきだ。

The individual tokens should be short as well. Perl and Common Lisp occupy opposite poles on this question. Perl programs can be almost cryptically dense, while the names of built-in Common Lisp operators are comically long. The designers of Common Lisp probably expected users to have text editors that would type these long names for them. But the cost of a long name is not just the cost of typing it. There is also the cost of reading it, and the cost of the space it takes up on your screen.

個々の語句も短いほうが良い。Perl と Common Lisp はこの点で両極端にある。Perl のプログラムはほとんど暗号とも呼べるほどに詰まっており、一方で Common Lisp の組み込みオペレータの名前は滑稽な程に長い。Common Lisp の設計者は、言語のユーザが名前を自動的に補完してくれるようなエディタを使うことを前提にしていたのかもしれない。しかし、長い名前が問題になるのは書くときだけじゃない。読むときにもコストがあるのだ。長い名前は画面上で大きなスペースを占有してしまう。

4 Hackability

4 ハックしやすさ

There is one thing more important than brevity to a hacker: being able to do what you want. In the history of programming languages a surprising amount of effort has gone into preventing programmers from doing things considered to be improper. This is a dangerously presumptuous plan. How can the language designer know what the programmer is going to need to do? I think language designers would do better to consider their target user to be a genius who will need to do things they never anticipated, rather than a bumbler who needs to be protected from himself. The bumbler will shoot himself in the foot anyway. You may save him from referring to variables in another package, but you can't save him from writing a badly designed program to solve the wrong problem, and taking forever to do it.

ハッカーにとって、簡潔さより大事なことがある。自分のやりたいことがやれることだ。プログラミング言語の歴史を振り返ってみると、「正しくない」と考えられる行いをプログラマがするのを防ぐために驚くべきほどの努力が払われてきた。これは危険なほどにおこ

がましい考えである。プログラマが必要とするであろうことを、どうやって言語設計者はあらかじめ知ることができるというのだ?言語設計者は、ユーザを自分のミスから守ってやらなきゃならないような間抜けではなく、設計者が考えもしなかったようなことを実現する必要に迫られている天才と考えた方が良いと思う。何をしようが間抜けは自分の足を撃つのだ。他のパッケージの変数を参照するミスを防いでやることはできたとしても、悪く設計されたプログラムを書くことから救うことはできない。

Good programmers often want to do dangerous and unsavory things. By unsavory I mean things that go behind whatever semantic facade the language is trying to present: getting hold of the internal representation of some high-level abstraction, for example. Hackers like to hack, and hacking means getting inside things and second guessing the original designer.

良いプログラマはしばしば危険で不道德なことをしたがる。不道德とは、ここでは言語が提示しようとしている意味構造の裏口を覗くような行為だ。高レベルの抽象化されたデータの内部表現をいじってみるようなことだ。ハッカーはハックするのが好きで、ハックというのはそもそもものごとの内側に入り込んで元の設計者を後知恵で批判するようなものだからだ。

Let yourself be second guessed. When you make any tool, people use it in ways you didn't intend, and this is especially true of a highly articulated tool like a programming language. Many a hacker will want to tweak your semantic model in a way that you never imagined. I say, let them; give the programmer access to as much internal stuff as you can without endangering runtime systems like the garbage collector.

後から批判され修正されることを恐れるな。どんなツールでも、人々は元の設計者が意図しなかったように使うものだし、プログラミング言語のように明確な意図のあるものでは特にそうだ。たくさんのハッカーが、あなたが想像だにしなかった方法で意味論的モデルをいじりたがるだろう。そうさせよう。ガベージコレクタのようなランタイムシステムを危険にさらさないぎりぎりのところまで、可能な限り、内部にアクセスできるようにしておこう。

In Common Lisp I have often wanted to iterate through the fields of a struct-- to comb out references to a deleted object, for example, or find fields that are uninitialized. I know the structs are just vectors underneath. And yet I can't write a general purpose function that I can call on any struct. I can only access the fields by name, because that's what a struct is supposed to mean.

私は **Common Lisp** でしばしば構造体のフィールドをループしたいと思うことがあった。例えば削除されたオブジェクトへの参照を除去したり、初期化されていないフィールドを見つけたりするようなことだ。構造体が内部では単なるベクターで表現されていることを私は知っていた。それでも私はどんな構造体に対しても呼べる汎用関数を書くことができない。構造体のフィールドには名前でしかアクセスできず、それがそもそも構造体の意図するところだからだ。

A hacker may only want to subvert the intended model of things once or twice in a big program. But what a difference it makes to be able to. And it may be more than a question of just solving a problem. There is a kind of pleasure here too. Hackers share the surgeon's secret pleasure in poking about in gross innards, the teenager's secret pleasure in popping zits. [2] For boys, at least, certain kinds of horrors are fascinating. Maxim magazine publishes an annual volume of photographs, containing a mix of pin-ups and grisly accidents. They know their audience.

ハッカーが大きなプログラムの中で意図されたモデルを破りたいと思うのはたぶん 1 箇所か 2 箇所くらいのものだ。それでも、それがどんなに大きな違いをもたらすことか。そして、それは単に問題を解決できるというだけじゃない。そこには一種の快感が潜んでいる。外科医がぞっとするような内臓をのぞき込む時の密かな快感や、ティーンエイジャーがニキビを潰す時の密かな快感と同じだ。[2] 少なくとも男の子にとって、ある種の恐怖は魅力的だ。雑誌『Maxim は』は毎年ピンナップとぞっとするような事故の写真を集めた写真集を発刊している。全く、読者をよく分かっている。

Historically, Lisp has been good at letting hackers have their way. The political correctness of Common Lisp is an aberration. Early Lisps let you get your hands on everything. A good deal of that spirit is, fortunately, preserved in macros. What a wonderful thing, to be able to make arbitrary transformations on the source code. 歴史的に、Lisp はハッカーたちに内部をいじくるらせるのがうまかった。Common Lisp の政治的に正しくあろうとする態度の方が邪道なのだ。初期の Lisp は何でもありだった。幸いなことに、マクロにはその精神の多くがまだ残されている。ソースコードを好きなように変換できるということのなんと素晴らしいことか。

Classic macros are a real hacker's tool-- simple, powerful, and dangerous. It's so easy to understand what they do: you call a function on the macro's arguments, and whatever it returns gets inserted in place of the macro call. Hygienic macros embody the opposite principle. They try to protect you from understanding what

they're doing. I have never heard hygienic macros explained in one sentence. And they are a classic example of the dangers of deciding what programmers are allowed to want. Hygienic macros are intended to protect me from variable capture, among other things, but variable capture is exactly what I want in some macros.

古典的なマクロは本物のハッカーの道具だ。簡単で、強力で、危険だ。それが何をするかを理解するのはあまりに容易だ。マクロの引数をもって関数を呼び出し、それが返したものをマクロ呼び出しがある場所に挿入する。健全なマクロは全く反対の原理を表明している。それがやっていることを理解するのを妨げているのだ。健全なマクロを一つの文で説明したものを見たことがない。健全なマクロはプログラマの望むところを規定しようとする危険な思想の古い例である。健全なマクロは、他のいくつかのこととともに、変数の捕捉から私を守ってくれるが、変数の捕捉こそが私がマクロでやりたいことなのだ。

A really good language should be both clean and dirty: cleanly designed, with a small core of well understood and highly orthogonal operators, but dirty in the sense that it lets hackers have their way with it. C is like this. So were the early Lisps. A real hacker's language will always have a slightly raffish character.

本当に良い言語は、綺麗でかつ汚くあるべきだ。綺麗にデザインされ、直交性の高いオペレータとよく理解された小さなコアで構成され、しかしハッカーが好きなことをできるような汚さを備えている。C がこれに当てはまる。初期の Lisp もそうだった。本当のハッカーの言語は多少野卑な性質を持っているものだ。

A good programming language should have features that make the kind of people who use the phrase "software engineering" shake their heads disapprovingly. At the other end of the continuum are languages like Ada and Pascal, models of propriety that are good for teaching and not much else.

良い言語は、「ソフトウェア工学」なんて言葉を使うような人々がこりやダメだと頭を振るような機能を持っているべきだ。その対極にある言語は Ada や Pascal のようなもので、礼儀正しくて教育には適しているがそれ以外のことにはあまり使えない。

5 Throwaway Programs

5 書き捨てのプログラム

To be attractive to hackers, a language must be good for writing the kinds of programs they want to write. And that means, perhaps surprisingly, that it has to be good for writing throwaway programs.

ハッカーを惹きつけるには、言語はハッカーが書きたがるようなプログラムを書くのに適していなければならない。それは、意外かもしれないが、書き捨てのプログラムを書くのに適していなければならないということだ。

A throwaway program is a program you write quickly for some limited task: a program to automate some system administration task, or generate test data for a simulation, or convert data from one format to another. The surprising thing about throwaway programs is that, like the "temporary" buildings built at so many American universities during World War II, they often don't get thrown away. Many evolve into real programs, with real features and real users.

書き捨てのプログラムとは、限定された仕事をするために手早く書き上げる類のプログラムのことだ。何かのシステム管理の仕事を自動化したり、シミュレーションのテストデータを生成したり、データのフォーマット変換をするような。このような書き捨てのプログラムに関する意外な事実とは、第二次大戦中に米国の多くの大学で建設された「一時的な」ビルディングのように、それがしばしば捨てられずに使われ続けることである。たくさんのそういったプログラムは本物のユーザと本物の機能を得て、本物のプログラムへと進化してゆく。

I have a hunch that the best big programs begin life this way, rather than being designed big from the start, like the Hoover Dam. It's terrifying to build something big from scratch. When people take on a project that's too big, they become overwhelmed. The project either gets bogged down, or the result is sterile and wooden: a shopping mall rather than a real downtown, Brasilia rather than Rome, Ada rather than C.

私は、最良の大きなプログラムは、フーバーダムのように最初から大きく設計されるのではなく、書き捨てのプログラムから生まれるんじゃないかという気がしている。ゼロから巨大なものを建設するのはおっかないことだ。巨大過ぎるプロジェクトに取り掛かるとき、人々は圧倒される。そんなプロジェクトは泥沼にはまるか、貧弱で気の抜けた結果しか出さない。本物のダウンタウンの代わりにショッピングモールになったり、ローマではなくブラジリアになったり、Cではなく Ada になったり。

Another way to get a big program is to start with a throwaway program and keep improving it. This approach is less daunting, and the design of the program benefits from evolution. I think, if one looked, that this would turn out to be the way most big programs were developed. And those that did evolve this way are probably still written in whatever language they were first written in, because it's

rare for a program to be ported, except for political reasons. And so, paradoxically, if you want to make a language that is used for big systems, you have to make it good for writing throwaway programs, because that's where big systems come from.

大きなプログラムを書く別の方法は、書き捨てるプログラムから始めて改善し続けることだ。このアプローチは挫折しづらいし、得られた知見をプログラムの設計に反映していくことができる。見回してみれば、多くの大きなプログラムはこのようにして作られてきた。そしてこの方法で進化を遂げたものは、依然としてそれが最初に書かれた言語で書かれているだろう。プログラムが他の言語に移植されるなんて、政治的な理由でもなけりゃ起こらないからだ。したがって、逆説的であるが、もし大きなシステムで使われる言語を作りたいと思うなら、書き捨てるプログラムを書くのに良い言語にしくちゃならない。大きなシステムはそこから始まるからだ。

Perl is a striking example of this idea. It was not only designed for writing throwaway programs, but was pretty much a throwaway program itself. Perl began life as a collection of utilities for generating reports, and only evolved into a programming language as the throwaway programs people wrote in it grew larger. It was not until Perl 5 (if then) that the language was suitable for writing serious programs, and yet it was already massively popular.

Perl がこのアイディアの最も印象的な例であろう。Perl は書き捨てるプログラムのために設計されたというだけでなく、Perl 自身が書き捨てるプログラムのようなものだった。Perl はレポートを生成するためのユーティリティのコレクションとして産声を上げ、人々が書く書き捨てるプログラムが大きくなってからプログラミング言語へと進化した。Perl 5 になってようやく重要なプログラムを書くのに相応しい言語になったが、既にその時には非常にポピュラーになっていたのだ。

What makes a language good for throwaway programs? To start with, it must be readily available. A throwaway program is something that you expect to write in an hour. So the language probably must already be installed on the computer you're using. It can't be something you have to install before you use it. It has to be there. C was there because it came with the operating system. Perl was there because it was originally a tool for system administrators, and yours had already installed it.

書き捨てるプログラムを書くのに良い言語とはどういうものだろう。まず、それはいつでも使える状態になければならない。書き捨てるプログラムとはあなたが 1 時間かそこいらで書きたいと思うものだ。だからその言語は既にあなたのマシンにインストールされてい

なければならないだろう。使う前にわざわざインストールしなくちゃならないんじゃないんだ。そこにある。Cはそこにある。OSに付いてくるからだ。Perlもそこにある。もともとシステム管理のためのツールで、それゆえに既にインストールされていることが多いからだ。

Being available means more than being installed, though. An interactive language, with a command-line interface, is more available than one that you have to compile and run separately. A popular programming language should be interactive, and start up fast.

しかし、使える状態にあるとは、インストールされているというだけではない。コマンドラインインタフェースを持つ会話的な言語のほうが、いちいちコンパイルと実行を分けなくちゃならない言語より使える。人気の出るプログラミング言語は会話的でなければならず、しかも素早く立ち上がらなくちゃならない。

Another thing you want in a throwaway program is brevity. Brevity is always attractive to hackers, and never more so than in a program they expect to turn out in an hour.

書き捨てのプログラムに必要なもうひとつのことは、簡潔さだ。簡潔さは常にハッカーを惹きつけるが、1時間で書き上げたいと思うようなプログラムではなおさらだ。

6 Libraries

6 ライブラリ

Of course the ultimate in brevity is to have the program already written for you, and merely to call it. And this brings us to what I think will be an increasingly important feature of programming languages: library functions. Perl wins because it has large libraries for manipulating strings. This class of library functions are especially important for throwaway programs, which are often originally written for converting or extracting data. Many Perl programs probably begin as just a couple library calls stuck together.

もちろん、究極の簡潔さとは、欲しいプログラムが既にかかれていてそれと呼ぶだけ、というものだ。そこで、私がプログラミング言語の中でますます重要になってゆくと思われることに話を移そう。ライブラリ関数だ。Perlは文字列操作の豊富なライブラリのおかげで人気が出た。この種のライブラリは書き捨てのプログラムでは特に重要だ。データを変換したり取り出したりするものが多いからだ。たくさんのPerlプログラムは、最初はいくつかのライブラリコールをつなぎ合わせただけのものから始まるだろう。

I think a lot of the advances that happen in programming languages in the next fifty years will have to do with library functions. I think future programming languages will have libraries that are as carefully designed as the core language. Programming language design will not be about whether to make your language strongly or weakly typed, or object oriented, or functional, or whatever, but about how to design great libraries. The kind of language designers who like to think about how to design type systems may shudder at this. It's almost like writing applications! Too bad. Languages are for programmers, and libraries are what programmers need.

この先 50 年のプログラミング言語の進化は、ライブラリ関数に関するものになるだろうと思う。未来のプログラミング言語は、言語の核と同じくらい慎重に設計されたライブラリを備えているだろう。プログラミング言語の設計の重点は、静的型付けか動的型付けかとか、オブジェクト指向か関数型かとかそういうことではなく、どうやったら素晴らしいライブラリを設計できるかということになってゆくだろう。型システムの設計みたいなことを考えるのが好きな言語設計者は身震いするかもしれない。ライブラリの設計だなんて、アプリケーションを書くみたいなことじゃないか！残念でした。言語はプログラマのためのもので、プログラマが必要としているのはライブラリなのだ。

It's hard to design good libraries. It's not simply a matter of writing a lot of code. Once the libraries get too big, it can sometimes take longer to find the function you need than to write the code yourself. Libraries need to be designed using a small set of orthogonal operators, just like the core language. It ought to be possible for the programmer to guess what library call will do what he needs.

良いライブラリを設計するのは難しい。たくさんのコードを書けばよいというものではない。ライブラリが大きくなり過ぎると、必要な関数を探し回るより自分で書いてしまったほうが早いなんてことが起こる。ライブラリは、コア言語と同様に、小さな数の直交性の高いオペレータを使って設計されなければならない。プログラマが、どのライブラリが欲しい機能を実行できるかを推測できるようになっていなければならない。

Libraries are one place Common Lisp falls short. There are only rudimentary libraries for manipulating strings, and almost none for talking to the operating system. For historical reasons, Common Lisp tries to pretend that the OS doesn't exist. And because you can't talk to the OS, you're unlikely to be able to write a serious program using only the built-in operators in Common Lisp. You have to use some implementation-specific hacks as well, and in practice these tend not to

give you everything you want. Hackers would think a lot more highly of Lisp if Common Lisp had powerful string libraries and good OS support.

ライブラリは、Common Lisp が失敗した点のひとつだ。文字列操作にはごく原始的なライブラリしかないし、オペレーティングシステムにアクセスするものはほとんど皆無だ。歴史的な理由から、Common Lisp は OS なんて存在しないかのように振る舞ってきた。だが OS にアクセスできないということは、Common Lisp 組み込みのオペレータだけではまともなプログラムは書けないということになる。実装に依存する何らかのハックをつかわなくちゃならないだろう。そして、そういうハックはあなたの欲しいもの全てを与えてはくれない。Common Lisp に強力な文字列ライブラリと良い OS インタフェースがあったなら、ハッカーは Lisp をもっと好んでいただろう。

7 Syntax

7 構文

Could a language with Lisp's syntax, or more precisely, lack of syntax, ever become popular? I don't know the answer to this question. I do think that syntax is not the main reason Lisp isn't currently popular. Common Lisp has worse problems than unfamiliar syntax. I know several programmers who are comfortable with prefix syntax and yet use Perl by default, because it has powerful string libraries and can talk to the os.

Lisp の構文を持つ言語、もっと正確に言えば構文を持たない言語はポピュラーになれるだろうか? この問題の答えを私は知らない。私が考えるのは、Lisp が現在ポピュラーでない主要な理由はその構文ではないということだ。Common Lisp は、見慣れない構文よりもっと悪い問題を抱えている。前置構文を使いこなせるプログラマでも、強力な文字列処理と OS インタフェースのために Perl のほうを使う、というプログラマを私は何人か知っている。

There are two possible problems with prefix notation: that it is unfamiliar to programmers, and that it is not dense enough. The conventional wisdom in the Lisp world is that the first problem is the real one. I'm not so sure. Yes, prefix notation makes ordinary programmers panic. But I don't think ordinary programmers' opinions matter. Languages become popular or unpopular based on what expert hackers think of them, and I think expert hackers might be able to deal with prefix notation. Perl syntax can be pretty incomprehensible, but that has not stood in the way of Perl's popularity. If anything it may have helped foster a Perl cult.

前置記法には 2 つばかり問題があるかもしれない。プログラマにとって馴染みが少ないということと、密度が高くないということだ。Lisp の世界の伝統的な解釈では最初の問題こそが真の問題とされていた。だが本当にそうだろうか。確かに前置記法は普通のプログラマを混乱させる。だが、普通のプログラマの意見は重要じゃない。優れたハッカーがどう考えるかが言語の人気を決めるのだ。Perl の構文はほとんど理解不能だが、それが Perl の人気の足を引っ張っているとは思えない。Perl が備えているそういう性質はむしろ Perl 信者を元気づけてきたのだ。

A more serious problem is the diffuseness of prefix notation. For expert hackers, that really is a problem. No one wants to write (aref a x y) when they could write a[x,y].

より深刻な問題は、前置記法の散漫さだ。優れたハッカーにとってはこれが本当の問題だ。誰も、a[x,y]と書けるところを(aref a x y)と書きたいとは思わない。

In this particular case there is a way to finesse our way out of the problem. If we treat data structures as if they were functions on indexes, we could write (a x y) instead, which is even shorter than the Perl form. Similar tricks may shorten other types of expressions.

この特定の例に関しては解決策がある。データ構造を、インデックスが渡された時は関数であるかのように振舞わせれば、(a x y)のように書くことができ、これは Perl の書式よりも短い。他の式も似たようなトリックを使って短くできるだろう。

We can get rid of (or make optional) a lot of parentheses by making indentation significant. That's how programmers read code anyway: when indentation says one thing and delimiters say another, we go by the indentation. Treating indentation as significant would eliminate this common source of bugs as well as making programs shorter.

インデントに意味を持たせれば、たくさんの括弧を取り除くか、もしくは省略可能にできる。プログラマがコードを読む時は結局インデントを見ているのだ。インデントの示す構造とデリミタの示す構造が食い違っている場合、私たちは大抵インデントのほうを見てしまう。インデントに意味を持たせれば、このようなよくあるバグの元を減らせるとともに、プログラムを短くできる。

Sometimes infix syntax is easier to read. This is especially true for math expressions. I've used Lisp my whole programming life and I still don't find prefix math expressions natural. And yet it is convenient, especially when you're

generating code, to have operators that take any number of arguments. So if we do have infix syntax, it should probably be implemented as some kind of read-macro.

中置記法は確かに読みやすい。とくに数式ではそうだ。私はプログラマのキャリアとしてずっと Lisp を使ってきたが、それでも前置記法による数式が自然であるとは思えない。だがそれは便利なのだ。特に可変長引数を取るオペレータを使うコードを生成するような場合にだ。もし中置記法を採用するなら、リーダーマクロの一種として実装するのがよいだろう。

I don't think we should be religiously opposed to introducing syntax into Lisp, as long as it translates in a well-understood way into underlying s-expressions. There is already a good deal of syntax in Lisp. It's not necessarily bad to introduce more, as long as no one is forced to use it. In Common Lisp, some delimiters are reserved for the language, suggesting that at least some of the designers intended to have more syntax in the future.

Lisp に構文を持ち込むことに対して宗教的な反発を感じるべきではないと思う。少なくともその構文が、ベースとなる S 式へとよく理解された方法で変換されるならば。既に Lisp には結構な量の構文が持ち込まれているのだし、その使用を強制しないのであればさらに構文を追加しても悪いことはなかろう。Common Lisp ではいくつかのデリミタは言語に予約となっているから、少なくとも設計者の一部は将来もっと多くの構文を追加することを意図していたのだろう。

One of the most egregiously unlispy pieces of syntax in Common Lisp occurs in format strings; format is a language in its own right, and that language is not Lisp. If there were a plan for introducing more syntax into Lisp, format specifiers might be able to be included in it. It would be a good thing if macros could generate format specifiers the way they generate any other kind of code.

Common Lisp の中で、言語道断に Lisp らしからぬ構文は format 文字列にある。format はそれ自身の言語を持っており、それは Lisp ではない。Lisp にさらに構文を追加するなら format 記述子もそれに含めるべきだ。マクロが他のコードを生成するのと同じように format 記述子も生成できたらよい。

An eminent Lisp hacker told me that his copy of CLTL falls open to the section format. Mine too. This probably indicates room for improvement. It may also mean that programs do a lot of I/O.

ある非常に優れた Lisp ハッカーが、彼の CLTL は format のところで割れてしまったと私

に語ったことがある。実は私のもそうだ。たぶんこれは改善すべき箇所を暗示している。また、プログラムは I/O をたくさんするものだというとも意味している。

8 Efficiency

8 効率

A good language, as everyone knows, should generate fast code. But in practice I don't think fast code comes primarily from things you do in the design of the language. As Knuth pointed out long ago, speed only matters in certain critical bottlenecks. And as many programmers have observed since, one is very often mistaken about where these bottlenecks are.

皆知っているように、良い言語は速いコードを生成しなければならない。しかし現実には、言語設計において速いコードを出すことが最優先されるべきとは思えない。クヌースがずっと昔に指摘したように、速度は重要なボトルネックとなる箇所のみで問題となる。そして、たくさんのプログラマが経験していることだが、どこがボトルネックかを読み違えるのは非常によくある間違いだ。

So, in practice, the way to get fast code is to have a very good profiler, rather than by, say, making the language strongly typed. You don't need to know the type of every argument in every call in the program. You do need to be able to declare the types of arguments in the bottlenecks. And even more, you need to be able to find out where the bottlenecks are.

したがって、速いコードを得る現実的な方法は、言語を静的型付けにしたりすることではなく、非常に良いプロファイラを作ることだ。プログラム中の全ての関数の全ての引数の型を知る必要なんてない。ただ、ボトルネックとなるところの引数の型を宣言してやる必要はある。そうするには、どこがボトルネックかを知る必要がある。

One complaint people have had with Lisp is that it's hard to tell what's expensive. This might be true. It might also be inevitable, if you want to have a very abstract language. And in any case I think good profiling would go a long way toward fixing the problem: you'd soon learn what was expensive.

Lisp に関して言われ続けてきた不満は、何が高価な操作かを知るのが難しいというものだ。これはおそらく真実だろう。だが、高度に抽象的な言語を持つ以上それは避け難いことでもある。そして、良いプロファイリングこそが問題を解決する正しい道への一歩だと私は考える。何が高価かはそれで知ることができる。

Part of the problem here is social. Language designers like to write fast compilers. That's how they measure their skill. They think of the profiler as an add-on, at best. But in practice a good profiler may do more to improve the speed of actual programs written in the language than a compiler that generates fast code. Here, again, language designers are somewhat out of touch with their users. They do a really good job of solving slightly the wrong problem.

この問題の一部は社会的なものだ。言語設計者は速いコンパイラを書きたがる。だってそれが彼らの技術を示すものだから。彼らにとっては、プロファイラは眼中にあったとしても附属品にすぎない。だが現場では、実際のプログラムを改善するには速いコードを生成するコンパイラよりも良いプロファイラのほうが役に立つ。ここでも、言語設計者はそのユーザから乖離している。彼らは問題を解決するために素晴らしい仕事をしているが、その問題は現場の問題とちょっとばかりずれているのだ。

It might be a good idea to have an active profiler-- to push performance data to the programmer instead of waiting for him to come asking for it. For example, the editor could display bottlenecks in red when the programmer edits the source code. Another approach would be to somehow represent what's happening in running programs. This would be an especially big win in server-based applications, where you have lots of running programs to look at. An active profiler could show graphically what's happening in memory as a program's running, or even make sounds that tell what's happening.

アクティブプロファイラ——プログラマが尋ねた時だけでなく、パフォーマンスデータを常にプログラマに示してくれるようなプロファイラ——は良いアイディアかもしれない。例えばプログラマがソースを編集するときに、エディタはボトルネックを赤く表示するといった具合だ。別のアプローチは、走っているプログラムの中で起こっていることを何らかの方法で表現することだ。サーバベースのアプリケーションではこれは特に役に立つだろう。たくさんの走っているプログラムを見ていなければならないからだ。アクティブプロファイラは走っているプログラムのメモリのアクセス状況をグラフィカルに表示したりとか、音で知らせるとかできるかもしれない。

Sound is a good cue to problems. In one place I worked, we had a big board of dials showing what was happening to our web servers. The hands were moved by little servomotors that made a slight noise when they turned. I couldn't see the board from my desk, but I found that I could tell immediately, by the sound, when there was a problem with a server.

音は問題を知らせる良い合図だ。私がかつて仕事をしたところでは、Web サーバの状態を

示すメーターのついた大きなボードがあった。何かあると、メーターの針が小さなサーボモーターで動かされるのでちょっとした音を立てる。私の席からはそのボードは見えなかったのだが、サーバに問題が生じた時は音だけですぐにわかった。

It might even be possible to write a profiler that would automatically detect inefficient algorithms. I would not be surprised if certain patterns of memory access turned out to be sure signs of bad algorithms. If there were a little guy running around inside the computer executing our programs, he would probably have as long and plaintive a tale to tell about his job as a federal government employee. I often have a feeling that I'm sending the processor on a lot of wild goose chases, but I've never had a good way to look at what it's doing.

効率の悪いアルゴリズムを自動的に検出するプロファイラを書くことさえも可能かもしれない。ある種のメモリアクセスのパターンが、悪いアルゴリズムのサインだと聞いても私は驚かない。コンピュータの中に我々のプログラムを実行してくれるこびとが走り回っているとしたら、彼は連邦政府で働く人が語るのと同じくらい、彼の仕事のどこがどれだけ非効率かを語ってくれるだろう。私はよく、ああプロセッサにさんざん無駄骨を折らせているんだろうなと感じることがあるが、実際に何がどうなっているのかを見る良い方法はなかった。

A number of Lisps now compile into byte code, which is then executed by an interpreter. This is usually done to make the implementation easier to port, but it could be a useful language feature. It might be a good idea to make the byte code an official part of the language, and to allow programmers to use inline byte code in bottlenecks. Then such optimizations would be portable too.

現代の多くの Lisp 処理系はバイトコードにコンパイルしてそれをインタプリタで実行する。これは移植を簡単にするために行われることが多いが、言語の機能としても有用かもしれない。バイトコードを言語のオフィシャルな定義の一部としてしまつて、プログラマがボトルネックの部分ではバイトコードをインラインで書けるようにするのだ。そうすればそういう最適化さえも移植可能になる。

The nature of speed, as perceived by the end-user, may be changing. With the rise of server-based applications, more and more programs may turn out to be i/o-bound. It will be worth making i/o fast. The language can help with straightforward measures like simple, fast, formatted output functions, and also with deep structural changes like caching and persistent objects.

エンドユーザが感じるアプリケーションのスピードの性質は変化してきている。サーバベ

ースのアプリケーションが増えるにつれ、プログラムは I/O に律速されることが多くなってきた。I/O を速くすることには価値があるだろう。言語は、単純にフォーマット出力を速くすることによっても、またキャッシュや永続オブジェクトのような深い内部構造の変更によってもこれをサポートすることができる。

Users are interested in response time. But another kind of efficiency will be increasingly important: the number of simultaneous users you can support per processor. Many of the interesting applications written in the near future will be server-based, and the number of users per server is the critical question for anyone hosting such applications. In the capital cost of a business offering a server-based application, this is the divisor.

ユーザはレスポンスタイムに関心がある。しかし、もうひとつの効率の指標が重要になってきている。プロセッサ当たりのサポート可能なユーザ数だ。近い将来、たくさんの興味深いアプリケーションがサーバベースで書かれるようになるだろう。そのようなアプリケーションをホストする立場で最も重要なのがサーバ当たりどれだけのユーザを処理できるかという点だ。サーバベースのビジネスの総コストに対して、その値は分母に効いてくる。

For years, efficiency hasn't mattered much in most end-user applications. Developers have been able to assume that each user would have an increasingly powerful processor sitting on their desk. And by Parkinson's Law, software has expanded to use the resources available. That will change with server-based applications. In that world, the hardware and software will be supplied together. For companies that offer server-based applications, it will make a very big difference to the bottom line how many users they can support per server.

長い間、エンドユーザ向けのアプリケーションにとって効率はあまり重要ではなかった。開発者は、いつでもユーザがより強力なプロセッサを手に入れていくことを当てにできた。そしてパーキンソンの法則により、ソフトウェアはそこにある資源を使い尽くすように拡大してきた。サーバベースのアプリケーションではこの流れは変わるだろう。そこでは、ハードウェアとソフトウェアはセットで提供される。サーバベースのアプリケーションを提供する企業は、サーバ当たりのユーザ数の違いが大きな収益の差につながるだろう。

In some applications, the processor will be the limiting factor, and execution speed will be the most important thing to optimize. But often memory will be the limit; the number of simultaneous users will be determined by the amount of memory you need for each user's data. The language can help here too. Good support for threads will enable all the users to share a single heap. It may also

help to have persistent objects and/or language level support for lazy loading.

ある種のアプリケーションでは、プロセッサが律速となり、実行速度の最適化が主要な課題となるだろう。だが、メモリが律速となることも多い。同時接続ユーザ数は各ユーザのために必要なメモリ量でも決ってくる。ここでも言語が助けになる可能性がある。良いスレッドのサポートがあれば、全てのユーザがひとつのヒープを共有することも可能かもしれない。永続オブジェクトのサポートや、言語レベルでの遅延ロードなども助けになるだろう。

9 Time

9 時間

The last ingredient a popular language needs is time. No one wants to write programs in a language that might go away, as so many programming languages do. So most hackers will tend to wait until a language has been around for a couple years before even considering using it.

人気のある言語が必要とする最後の要素は時間である。誰も、消えてしまう言語でプログラムを書きたいとは思わない。そしてたくさんの言語が現われては消えていく。たいていのハッカーは、言語が作られてから最低でも 2 年間くらい経ってようやく、それを使うことを考え始める。

Inventors of wonderful new things are often surprised to discover this, but you need time to get any message through to people. A friend of mine rarely does anything the first time someone asks him. He knows that people sometimes ask for things that they turn out not to want. To avoid wasting his time, he waits till the third or fourth time he's asked to do something; by then, whoever's asking him may be fairly annoyed, but at least they probably really do want whatever they're asking for.

新しい素敵なものを考え出す発明家はしばしば見落とすのだが、人々にメッセージが行き渡るには時間がかかるのだ。私のある友人は誰かに何かを頼まれても、1 回目は何もしようとしない。人々はよく何かを思い付きで頼んで、後からそれが不要だったと分かることが多いと彼は知っているからだ。時間を浪費しないために、彼は 3 回か 4 回目のリクエストが来るまで待つ。そのころにはリクエストを出した人はかなりいらっているが、少なくともその人はそれが本当に必要だということが分かっている。

Most people have learned to do a similar sort of filtering on new things they hear about. They don't even start paying attention until they've heard about something

ten times. They're perfectly justified: the majority of hot new whatever's do turn out to be a waste of time, and eventually go away. By delaying learning VRML, I avoided having to learn it at all.

たいていの人は、耳にする新しいことに対して、似たようなフィルタを身に付けている。何かを 10 回は耳にするまで、人々はそのことに注意を払おうとさえもしない。これは全く正当なことだ。新しくてホットな何とやらの大部分は時間の無駄で、消えていってしまうものだからだ。私自身、VRML を学ぶのを遅らせることでそれを学ぶことそのものを避けることができた。

So anyone who invents something new has to expect to keep repeating their message for years before people will start to get it. We wrote what was, as far as I know, the first web-server based application, and it took us years to get it through to people that it didn't have to be downloaded. It wasn't that they were stupid. They just had us tuned out.

したがって、何か新しいものを発明した人は、人々がその価値に気付くまで、何年も根気良くメッセージを繰り返すことを覚悟しておかねばならない。私たちは知る限りで最初のサーバベースアプリケーションを書いたが、それを使うのに何もダウンロードする必要がないと人々に納得させるのに何年もかかったのだ。彼らが間抜けだったわけではない。彼らは私たちにそのことに気付かせたのだ。

The good news is, simple repetition solves the problem. All you have to do is keep telling your story, and eventually people will start to hear. It's not when people notice you're there that they pay attention; it's when they notice you're still there. この事実の良い面は、単純な繰り返しで解決できることだ。とにかく自分の伝えたいことを話し続ければ、いずれ人々は耳を傾けてくれるようになる。人々があなたに注意を払うのは、あなたがそこにいることに気付いた時じゃなく、あなたがまだそこにいることに気付いた時だ。

It's just as well that it usually takes a while to gain momentum. Most technologies evolve a good deal even after they're first launched-- programming languages especially. Nothing could be better, for a new technology, than a few years of being used only by a small number of early adopters. Early adopters are sophisticated and demanding, and quickly flush out whatever flaws remain in your technology. When you only have a few users you can be in close contact with all of them. And early adopters are forgiving when you improve your system, even if this causes some breakage.

また、勢いを得るまでにも時間がかかるのが普通だ。多くの技術は最初に発表された時から大きく発展を遂げるものだし、プログラミング言語では特にそうだ。新しい技術にとって一番良いことは、最初に採用してくれる少数の人々によって 2〜3 年間使われることだ。最初に使ってくれる人々は知識があり要求も高く、あなたの技術に残っている欠点を速やかに洗い流してくれる。また、ユーザが少数であればあなたは全員と緊密にコンタクトが取れる。最初に使ってくれる人たちは、あなたがシステムを改善することには、それが既存のシステムを壊すものであったとしても寛容だ。

There are two ways new technology gets introduced: the organic growth method, and the big bang method. The organic growth method is exemplified by the classic seat-of-the-pants underfunded garage startup. A couple guys, working in obscurity, develop some new technology. They launch it with no marketing and initially have only a few (fanatically devoted) users. They continue to improve the technology, and meanwhile their user base grows by word of mouth. Before they know it, they're big.

新しい技術を紹介するには 2 つの方法がある。有機的に成長する方法と、ビッグバンみたいな方法だ。有機的な成長は、低予算のガレージベンチャーに実例を見ることができる。誰にも知られずひっそりと、2 人ばかりの若者が、新しい技術を創り出す。彼らはそれをマーケティングもなしに発表し、最初はごく少数のユーザしか得られない(でもそのユーザはその技術の熱狂的なファンだ)。彼らは技術を改善し続け、同時にユーザベースが口コミで広がっていく。彼らがそれと気付く前に、彼らは大きくなっている。

The other approach, the big bang method, is exemplified by the VC-backed, heavily marketed startup. They rush to develop a product, launch it with great publicity, and immediately (they hope) have a large user base.

もう一つのアプローチ、ビッグバン方法は、ベンチャーキャピタルにバックアップされて、派手にマーケティングされたベンチャー企業に例を見ることができる。彼らは製品を出すのを急ぎ、華々しくそれを発表して、(彼らの望みでは)直ちに大きなユーザベースを得る。

Generally, the garage guys envy the big bang guys. The big bang guys are smooth and confident and respected by the VCs. They can afford the best of everything, and the PR campaign surrounding the launch has the side effect of making them celebrities. The organic growth guys, sitting in their garage, feel poor and unloved. And yet I think they are often mistaken to feel sorry for themselves. Organic growth seems to yield better technology and richer founders than the big bang method. If you look at the dominant technologies today, you'll find that most of

them grew organically.

だいたい、ガレー組はビックバン組を妬む。ビックバン組は如才なく、自信たっぷりを出資者からも一目置かれている。彼らは何でも最上の物を手に入れられ、発表の際の広告キャンペーンによって一躍有名人だ。有機的成長組はガレーに座って、貧しく愛されていないと感じる。だが、有機的成長組は自分のことをそんなふうを考えなくてもよいのだ。有機的成長はビックバンよりも結果的に良い技術と支持者を得ている。今日有力な技術を見れば、それらの多くは有機的に成長したものだと分かるだろう。

This pattern doesn't only apply to companies. You see it in sponsored research too. Multics and Common Lisp were big-bang projects, and Unix and MacLisp were organic growth projects.

このパターンは企業だけでなく、後援された研究にも当てはまる。Multics と Common Lisp はビッグバンプロジェクトで、Unix と MacLisp は有機的成長プロジェクトだった。

10 Redesign

10 再デザイン

"The best writing is rewriting," wrote E. B. White. Every good writer knows this, and it's true for software too. The most important part of design is redesign. Programming languages, especially, don't get redesigned enough.

「最も良い書き方は書き直すことだ(The best writing is rewriting)」と E.B.ホワイトは書いた。良い作家は皆これを知っている。ソフトウェアについてもこの命題は真実だ。デザインの最も重要な段階は再デザインである。プログラミング言語は特に十分に再デザインされていない。

To write good software you must simultaneously keep two opposing ideas in your head. You need the young hacker's naive faith in his abilities, and at the same time the veteran's skepticism. You have to be able to think [how hard can it be?](#) with one half of your brain while thinking [it will never work](#) with the other.

良いソフトウェアを書くには、2つの相反する考えを頭に置いておかななくてはならない。若いハッカーの無邪気な信頼と、ベテランの懐疑主義だ。「こんなの簡単だよ」と頭の半分で考える一方で、「こんなのうまくいきっこない」ともう半分で考えるのだ。

The trick is to realize that there's no real contradiction here. You want to be optimistic and skeptical about two different things. You have to be optimistic about the possibility of solving the problem, but skeptical about the value of

whatever solution you've got so far.

ポイントは、実はこの 2 つには矛盾はないということだ。あなたは異なるものについてそれぞれ楽観的と懐疑的になっているのだ。問題を解決できる可能性に関する楽観と、得られた解決の価値に対する懐疑だ。

People who do good work often think that whatever they're working on is no good. Others see what they've done and are full of wonder, but the creator is full of worry. This pattern is no coincidence: it is the worry that made the work good.

素晴らしい仕事をする人はしばしば、自分がやっていることには価値がないと考えてしまう。他の人が見ればその仕事はまさに驚異的なのに、それを創った当人には欠陥ばかりが目につく。だが、その現象は偶然ではない。その心配こそが、仕事を良くするものなのだ。

If you can keep hope and worry balanced, they will drive a project forward the same way your two legs drive a bicycle forward. In the first phase of the two-cycle innovation engine, you work furiously on some problem, inspired by your confidence that you'll be able to solve it. In the second phase, you look at what you've done in the cold light of morning, and see all its flaws very clearly. But as long as your critical spirit doesn't outweigh your hope, you'll be able to look at your admittedly incomplete system, and think, how hard can it be to get the rest of the way?, thereby continuing the cycle.

希望と心配のバランスをうまく取れれば、それはあなたが 2 本の足で自転車を漕ぐように、プロジェクトを前へ押し進めてくれる。イノベーションの 2 サイクルエンジンを考えてみよう。最初のフェーズでは、自分はその問題を解決できるという確信に押されて、あなたは一心不乱に問題に取り組む。第二フェーズでは、あなたは自分のやったことを冷たい朝日の下で眺め、その欠陥をはっきりと目にする。だが批判的な精神が希望を押し潰さない限り、あなたは自分のシステムが不完全であることを認め、考えるのだ。あとこんだけ頑張ればできるかな。そしてサイクルを続けるのだ。

It's tricky to keep the two forces balanced. In young hackers, optimism predominates. They produce something, are convinced it's great, and never improve it. In old hackers, skepticism predominates, and they won't even dare to take on ambitious projects.

2 つの力をバランスさせておくのは難しい。若いハッカーでは楽観主義が支配しがちだ。彼らは何かを創り出し、それが素晴らしいものだと思っていて疑わず、より良くしていこうと思わない。年老いたハッカーでは懐疑主義が支配し、野心的なプロジェクトに手を付けようとさえしない。

Anything you can do to keep the redesign cycle going is good. Prose can be rewritten over and over until you're happy with it. But software, as a rule, doesn't get redesigned enough. Prose has readers, but software has *users*. If a writer rewrites an essay, people who read the old version are unlikely to complain that their thoughts have been broken by some newly introduced incompatibility.

再デザインのサイクルを回し続けられるものなら何であれ良い。散文は、作者が満足するまで何度も何度も書き直される。しかしソフトウェアは一般的に十分に再デザインされない。散文には読者がいるが、ソフトウェアにいるのはユーザだ。作家がエッセイを書き直したところで、古い版を読んでいた読者が新しい版は前と互換性がなくなったから自分の考えが動かなくなったなんて文句を言うてくることはない。

Users are a double-edged sword. They can help you improve your language, but they can also deter you from improving it. So choose your users carefully, and be slow to grow their number. Having users is like optimization: the wise course is to delay it. Also, as a general rule, you can at any given time get away with changing more than you think. Introducing change is like pulling off a bandage: the pain is a memory almost as soon as you feel it.

ユーザは諸刃の剣だ。彼らはあなたの言語を改善してゆく助けになるが、同時に改善を止める要素にもなる。だからユーザは慎重に選ばなければならず、ゆっくりと増やしていかねばならない。ユーザを持つことは最適化のプロセスとも似ている。賢いやり方は、それを後回しにすることだ。また、一般的なルールとして、どんな時でも、思った以上に変更はきくものだ。変更を入れることは、包帯をはがすのに似ている。痛みを感じたと思ったらそれはもう過去の思い出だ。

Everyone knows that it's not a good idea to have a language designed by a committee. Committees yield bad design. But I think the worst danger of committees is that they interfere with redesign. It is so much work to introduce changes that no one wants to bother. Whatever a committee decides tends to stay that way, even if most of the members don't like it.

誰もが、言語は委員会で設計してはいけないと知っている。委員会は悪い設計を生む。だが私の意見では、委員会の最大の危険は、再デザインをやりにくくすることだ。誰もが現状でどうにか満足している時に新しい変更を入れるのには莫大な労力を要する。委員会が決めたことは、たとえ大部分のメンバーがそれを本当は気に入ってないにしても、そのままになりがちだ。

Even a committee of two gets in the way of redesign. This happens particularly in the interfaces between pieces of software written by two different people. To change the interface both have to agree to change it at once. And so interfaces tend not to change at all, which is a problem because they tend to be one of the most ad hoc parts of any system.

たった 2 人からなる委員会でさえ再デザインの妨げになる。別々の人が書いた 2 つのソフトウェア部品のインタフェースでよくそれは起こる。インタフェースを変更するには、両者がそれに合意して同時に変更しなければならない。そのせいで、インタフェースは変更されずに残ることが多く、結果としてどんなシステムでもやっつけ作業みたいな部分として抱えることになる。

One solution here might be to design systems so that interfaces are horizontal instead of vertical-- so that modules are always vertically stacked strata of abstraction. Then the interface will tend to be owned by one of them. The lower of two levels will either be a language in which the upper is written, in which case the lower level will own the interface, or it will be a slave, in which case the interface can be dictated by the upper level.

ひとつの解決法は、システムのインタフェースを左右方向でなく上下方向に設計することだ。モジュールが常に抽象化の階層として上下に積まれるようにする。そうするとインタフェースはモジュールのどちらかが責任を持つことになりやすい。下のモジュールが、上のモジュールが利用する言語だとすれば、下のモジュールがインタフェースに責任を持てるし、下のモジュールがスレーブであれば、上のモジュールがインタフェースを支配できる。

11 Lisp

11 Lisp

What all this implies is that there is hope for a new Lisp. There is hope for any language that gives hackers what they want, including Lisp. I think we may have made a mistake in thinking that hackers are turned off by Lisp's strangeness. This comforting illusion may have prevented us from seeing the real problem with Lisp, or at least Common Lisp, which is that it sucks for doing what hackers want to do. A hacker's language needs powerful libraries and something to hack. Common Lisp has neither. A hacker's language is terse and hackable. Common Lisp is not. ここまでの議論が暗示しているのは、新しい Lisp に希望があるということだ。ハッカーが欲しがるものを与える言語なら何でも希望があるし、Lisp もそうだ。Lisp の奇妙さをハッ

カーが嫌ったと結論づけるのは間違いだったんじゃないか。この淡い期待は、私たちの目を **Lisp** の本当の問題から逸らしてきた。少なくともハッカーがやりたいことを駄目にする **Common Lisp** についてはそうだ。ハッカーの言語は強力なライブラリとハックすべき物を必要とする。 **Common Lisp** にはどちらも無い。ハッカーの言語は簡潔でハック可能でなくてはならない。 **Common Lisp** はそうではない。

The good news is, it's not Lisp that sucks, but Common Lisp. If we can develop a new Lisp that is a real hacker's language, I think hackers will use it. They will use whatever language does the job. All we have to do is make sure this new Lisp does some important job better than other languages.

良いニュースもある。それは **Common Lisp** が駄目でも、**Lisp** は駄目じゃないってことだ。本物のハッカーの言語となるべき新しい **Lisp** を作ることができれば、ハッカーはそれを使うんじゃないかと私は思う。ハッカーはやりたいことにちゃんと使える言語なら何であれ使うだろう。私たちのやるべきことは、新しい **Lisp** がいくつかの重要な用途で他の言語よりもうまく動くようにすることだ。

History offers some encouragement. Over time, successive new programming languages have taken more and more features from Lisp. There is no longer much left to copy before the language you've made is Lisp. The latest hot language, Python, is a watered-down Lisp with infix syntax and no macros. A new Lisp would be a natural step in this progression.

歴史が勇気を与えてくれる。次々と現われる新しいプログラミング言語は、**Lisp** から機能を次々と採り入れている。新しい言語が **Lisp** になる前に欠けている機能はもうあまり残っていない。最も新しいホットな言語 **Python** は中置記法を使い、マクロを除いて **Lisp** を弱めたものだ。新しい **Lisp** は進歩における自然な一歩になるだろう。

I sometimes think that it would be a good marketing trick to call it an improved version of Python. That sounds hipper than Lisp. To many people, Lisp is a slow AI language with a lot of parentheses. Fritz Kunze's official biography carefully avoids mentioning the L-word. But my guess is that we shouldn't be afraid to call the new Lisp Lisp. Lisp still has a lot of latent respect among the very best hackers-- the ones who took 6.001 and understood it, for example. And those are the users you need to win.

マーケティングのためには、新しい **Lisp** を進歩した **Python** だと呼んだほうがいいんじゃないかと考えることもある。 **Lisp** とか言うよりもずっとカッコいい。多くの人にとって、**Lisp** は括弧だらけの遅い **AI** 向け言語だ。フリッツ・クンツェの公式な経歴からは **L** の付く

単語が慎重に除かれている。だが、新しい Lisp を Lisp と呼ぶことを恐れてはならないだろう。Lisp はいまだに、最高のハッカーたち——例えばコース 6.001 を履修して理解したような人たちから隠れた尊敬を集めている。彼らこそが私たちが惹き付けなければならない人々だ。

In "How to Become a Hacker," Eric Raymond describes Lisp as something like Latin or Greek-- a language you should learn as an intellectual exercise, even though you won't actually use it:

「ハッカーになるには」の中で、エリック・レイモンドは Lisp をラテン語かギリシャ語のように形容している。実際には使わないかもしれないが、教養のために学ぶべきだと。

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

LISP は、それをものにしたときのすばらしい悟り体験のために勉強しましょう。この体験は、その後の人生でよりよいプログラマーとなる手助けとなるはずです。たとえ、実際には LISP そのものをあまり使わなくても。

If I didn't know Lisp, reading this would set me asking questions. A language that would make me a better programmer, if it means anything at all, means a language that would be better for programming. And that is in fact the implication of what Eric is saying.

私が Lisp を知らずにこれを読んだとしたら、質問したくて仕方なくなっただろう。プログラミング言語が自分をよいプログラマにしてくれるってことは、それが何であろうと、その言語がプログラミングに優れているってことじゃないのか。実際、レイモンドの言っていることはそれを暗示している。

As long as that idea is still floating around, I think hackers will be receptive enough to a new Lisp, even if it is called Lisp. But this Lisp must be a hacker's language, like the classic Lisps of the 1970s. It must be terse, simple, and hackable. And it must have powerful libraries for doing what hackers want to do now.

こんな風に思われている限りにおいては、ハッカーは新しい Lisp を、たとえ Lisp と名がついていても受け入れるんじゃないかと思う。ただし、この Lisp はハッカーの言語じゃなくちゃだめだ。1970 年代の Lisp がそうであったように。簡潔で、単純で、ハック可能な。

そして、今日のハッカーがやりたいようなことを実現する強力なライブラリを備えていないくちやいけない。

In the matter of libraries I think there is room to beat languages like Perl and Python at their own game. A lot of the new applications that will need to be written in the coming years will be [server-based applications](#). There's no reason a new Lisp shouldn't have string libraries as good as Perl, and if this new Lisp also had powerful libraries for server-based applications, it could be very popular. Real hackers won't turn up their noses at a new tool that will let them solve hard problems with a few library calls. Remember, hackers are lazy.

ライブラリに関して言えば、Perl や Python といった言語に対して彼らの土俵で勝てる余地はあると私は思っている。これから書かれる新しいアプリケーションの多くはサーバベースのものになるだろう。だから、新しい Lisp が Perl と同じくらい良い文字列ライブラリを備え、さらにサーバベースのアプリケーションに必要な強力なライブラリを備えていたら、人気が出るんじゃないか。本物のハッカーは、難しい問題をほんの数個のライブラリ呼び出しで解決してくれるようなツールを軽蔑したりはしない。そう、ハッカーは怠惰なものなんだ。

It could be an even bigger win to have core language support for server-based applications. For example, explicit support for programs with multiple users, or data ownership at the level of type tags.

コア言語がサーバベースのアプリケーションをサポートしていればなお良いだろう。例えば、マルチユーザのプログラムを明示的にサポートしたり、タイプタグのレベルでデータの所有権をサポートしたりといったことだ。

Server-based applications also give us the answer to the question of what this new Lisp will be used to hack. It would not hurt to make Lisp better as a scripting language for Unix. (It would be hard to make it worse.) But I think there are areas where existing languages would be easier to beat. I think it might be better to follow the model of Tcl, and supply the Lisp together with a complete system for supporting server-based applications. Lisp is a natural fit for server-based applications. Lexical closures provide a way to get the effect of subroutines when the ui is just a series of web pages. S-expressions map nicely onto html, and macros are good at generating it. There need to be better tools for writing server-based applications, and there needs to be a new Lisp, and the two would work very well together.

サーバベースのアプリケーションはまた、新しい **Lisp** で何をハックすべきかという問いに答えてくれる。**Unix** のスクリプト言語とするだけでも **Lisp** は良いものになるだろうが(これ以上悪くするほうが難しい)、既存の言語と競争するのに勝てる領域というのはあると思うのだ。**Tcl** のモデルを真似るのがいいんじゃないか。サーバベースのアプリケーションをサポートする完全なシステムを作り、**Lisp** をそれと一緒に提供するのだ。**Lisp** はサーバベースアプリケーションに自然な形でフィットする。レキシカルクロージャは、**UI** が単に一連の **Web** ページである場合にサブルーチンの機能を提供できるし、**S** 式は **HTML** に綺麗にマッピングされるし、それを生成するのにマクロは最適だ。サーバベースのアプリケーションを書くのに良いツールが必要とされており、一方で新しい **Lisp** も必要とされている。この両者を一緒にやればうまくいくんじゃないか。

12 The Dream Language

12 夢の言語

By way of summary, let's try describing the hacker's dream language. The dream language is [beautiful](#), clean, and terse. It has an interactive toplevel that starts up fast. You can write programs to solve common problems with very little code. Nearly all the code in any program you write is code that's specific to your application. Everything else has been done for you.

まとめの代わりに、ハッカーの夢の言語というのを考えてみよう。夢の言語は美しく、明快で、簡潔だ。素早く立ち上がる会話的なトップレベルを持つ。よくある問題を解くプログラムはほんのわずかなコードで書ける。どんなプログラムでも、書かなくちゃいけないのはアプリケーションに特有な部分だけだ。他の全てはあなたのために既に作られている。

The syntax of the language is brief to a fault. You never have to type an unnecessary character, or even to use the shift key much.

その言語の構文は極端に短い。必要のない文字をタイプする必要はほとんどなく、シフトキーさえほとんど使わない。

Using big abstractions you can write the first version of a program very quickly. Later, when you want to optimize, there's a really good profiler that tells you where to focus your attention. You can make inner loops blindingly fast, even writing inline byte code if you need to.

高度な抽象化を使って最初のバージョンは極めて速く書き上げられる。後に、最適化しなくなったら、非常に良いプロファイラが提供されていて、どこに注目したらいいかを教えてくれる。必要とあらばインラインでバイトコードを書くことさえできて、内部のループ

を目も眩むほど速くすることができる。

There are lots of good examples to learn from, and the language is intuitive enough that you can learn how to use it from examples in a couple minutes. You don't need to look in the manual much. The manual is thin, and has few warnings and qualifications.

学ぶためにたくさんの良い例が提供されていて、さらに言語は直観的なので例からほんの 1～2 分で使い方を学ぶことができる。マニュアルを見る必要さえあまりない。マニュアルは薄く、注意すべき点や制限はほとんどない。

The language has a small core, and powerful, highly orthogonal libraries that are as carefully designed as the core language. The libraries all work well together; everything in the language fits together like the parts in a fine camera. Nothing is deprecated, or retained for compatibility. The source code of all the libraries is readily available. It's easy to talk to the operating system and to applications written in other languages.

その言語は小さなコアと、コアと同じくらい慎重に設計された強力な直交性の高いライブラリからなる。ライブラリはお互いがうまく協調して動く。良いカメラのように、言語の全ての部分はぴったりとかみ合って動くのだ。非推奨機能とか、互換性のために残された機能なんてものはない。ライブラリのソースコードも用意されている。オペレーティングシステムや他の言語で書かれたアプリケーションとおしゃべりするのも簡単だ。

The language is built in layers. The higher-level abstractions are built in a very transparent way out of lower-level abstractions, which you can get hold of if you want.

その言語はいくつもの層から構成されている。抽象化の高い層は抽象化の低い層から透過的に構成されており、必要とあらば抽象化の低い層を直接叩くこともできる。

Nothing is hidden from you that doesn't absolutely have to be. The language offers abstractions only as a way of saving you work, rather than as a way of telling you what to do. In fact, the language encourages you to be an equal participant in its design. You can change everything about it, including even its syntax, and anything you write has, as much as possible, the same status as what comes predefined.

絶対に必要である箇所以外は、何事もプログラマから隠されてはいない。その言語の抽象化機能は、プログラマにどうプログラムすべきかを教えるのではなく、プログラマの仕事

を楽にするためだけに提供されている。実際、その言語はプログラマを言語の設計に参加するものとして扱ってくれるのだ。プログラマは、言語そのものを、構文でさえも変更することができ、プログラマが書いたものは可能な限り、言語組み込みのものと同様の地位を得る。

Notes

注釈

[1] Macros very close to the modern idea were proposed by Timothy Hart in 1964, two years after Lisp 1.5 was released. What was missing, initially, were ways to avoid variable capture and multiple evaluation; Hart's examples are subject to both.

[1] 現在のものに極めて近いマクロは、Lisp 1.5 がリリースされた 2 年後、1964 年にティモシー・ハートによって提案された。欠けていたのは変数の捕捉を避ける方法と、複数回の評価を避ける方法であった。ハートの例はその両方だ。

[2] In *When the Air Hits Your Brain*, neurosurgeon Frank Vertosick recounts a conversation in which his chief resident, Gary, talks about the difference between surgeons and internists ("fleas"):

[2] ” *When the Air Hits Your Brain* ” の中で、神経外科医 Frank Vertosick は、彼の病棟医長である Gary と外科医とインターン(「蚤」)の違いについて交わした会話について語っている。

Gary and I ordered a large pizza and found an open booth. The chief lit a cigarette. "Look at those goddamn fleas, jabbering about some disease they'll see once in their lifetimes. That's the trouble with fleas, they only like the bizarre stuff. They hate their bread and butter cases. That's the difference between us and the fucking fleas. See, we love big juicy lumbar disc herniations, but they hate hypertension...."

Gary と私は大きなピザを頼んで空いている席を見つけた。医長は煙草に火をつけると言った。「あの蚤どもを見なよ。一生に一度お目にかかれるかどうかみたいな疾病のことばかり話している。それがあいつらの問題なんだ。珍しいものにしか興味がありやしない。それで食ってくような、日ごろよくある症例は嫌っているのさ。それが俺たちとあの蚤どもとの違いだ。俺たちはでかくて汁もしたたるような椎間板ヘルニアが大好物だが、やつらは高血圧でさえ嫌うのさ…

It's hard to think of a lumbar disc herniation as juicy (except literally). And yet I think I know what they mean. I've often had a juicy bug to track down. Someone who's not a programmer would find it hard to imagine that there could be pleasure in a bug. Surely it's better if everything just works. In one way, it is. And yet there is undeniably a grim satisfaction in hunting down certain sorts of bugs.

椎間板ヘルニアを(文字通りの意味でなく)汁もしたたるようなと形容する感覚を理解するのは難しいが、彼らの言わんとしているところが私には分かると思う。私もしばしば汁もしたたるようなバグを追いかけることがある。プログラマでない人は、いったいバグにどんな喜びがあるのか不思議でしょうがないと思う。そりゃもちろん全てがうまく動いてくれるに越したことはない。だが、ある種のバグを追いかけて潰した時には確かにぞくぞくするような満足感があるものなのだ。

Chapter 15 DESIGN AND RESEARCH

第 15 章 デザインとリサーチ

January 2003

(This article is derived from a keynote talk at the fall 2002 meeting of NEPLS.)

(この記事は 2002 年秋の NEPLS のミーティングの基調講演を基に作成されたものである。)

Visitors to this country are often surprised to find that Americans like to begin a conversation by asking "what do you do?" I've never liked this question. I've rarely had a neat answer to it. But I think I have finally solved the problem. Now, when someone asks me what I do, I look them straight in the eye and say "I'm designing a [new dialect of Lisp](#)." I recommend this answer to anyone who doesn't like being asked what they do. The conversation will turn immediately to other topics.

外国人はよく驚くのだが、米国人は会話の初めに「どんなお仕事をなさっていますか」と尋ねることがとても多い。私はこの質問が苦手だ。うまい答えを思い付いたためしがないんだ。でも最近、ついにこの問題は解決をみた。誰かが私の仕事を尋ねたら、その人の目をまっすぐ見て「Lisp の新しい方言を設計しています」と言うことにしたんだ。この答えは仕事を聞かれるのが苦手な人にはお勧めだよ。会話はあっという間に別の話題に移るだろうからね。

I don't consider myself to be doing research on programming languages. I'm just designing one, in the same way that someone might design a building or a chair or a new typeface. I'm not trying to discover anything new. I just want to make a language that will be good to program in. In some ways, this assumption makes life a lot easier.

私は、自分がプログラミング言語のリサーチをしているとは思っていない。人が、建物や椅子や新しいフォントをデザインするのと同じように、言語をデザインしているだけだ。何か新しいことを発見しようとしているわけじゃない。ただ、プログラムするのに良い言語を作りたいと思っている。こう考えることで、問題はずいぶん扱いやすくなっているようだ。

The difference between design and research seems to be a question of new versus good. Design doesn't have to be new, but it has to be good. Research doesn't have

to be good, but it has to be new. I think these two paths converge at the top: the best design surpasses its predecessors by using new ideas, and the best research solves problems that are not only new, but actually worth solving. So ultimately we're aiming for the same destination, just approaching it from different directions.

デザインとリサーチの違いは、新しさ対良さの問題だと言えると思う。デザインは必ずしも新しくある必要はないが、良くなくてはならない。リサーチは必ずしも良くある必要はないが、新しくなくてはならない。この 2 つの道は頂上で一緒になると私は思う。最高のデザインは新しいアイデアでもってそれまでのものを凌駕するだろうし、最高のリサーチは新しいだけでなく、解くに値する問題を解くのだ。したがって私たちは究極的には同じ方向を目指しているが、別々の方向から近づいていると言えるだろう。

What I'm going to talk about today is what your target looks like from the back. What do you do differently when you treat programming languages as a design problem instead of a research topic?

今日私が話したいのは、目標を反対側から眺めたらどんなふうに見えるかってことだ。プログラミング言語をリサーチの対象としてでなく、デザインの問題だととらえた場合、あなたはどんなふうにそれに取り組むだろう。

The biggest difference is that you focus more on the user. Design begins by asking, who is this for and what do they need from it? A good architect, for example, does not begin by creating a design that he then imposes on the users, but by studying the intended users and figuring out what they need.

一番わかりやすい違いは、よりユーザに注意を向けることだろう。これは誰のためで、どんな必要を満たすモノか、と問うところからデザインは始まる。例えば、良い建築家は、デザインを先に作ってから住む人に押しつけるのではなく、住む対象となる人を研究して、それらの人々が必要としているものを見つけることから始めるだろう。

Notice I said "what they need," not "what they want." I don't mean to give the impression that working as a designer means working as a sort of short-order cook, making whatever the client tells you to. This varies from field to field in the arts, but I don't think there is any field in which the best work is done by the people who just make exactly what the customers tell them to.

私が、「欲しているもの」ではなく「必要としているもの」と言ったことに注意して欲しい。

デザイナーの仕事は、お客の言う通りなんでもすぐ作ってみせる料理人みたいなものじゃない。芸術の分野によって違いはあるだろうが、どんな分野にせよ、最高の仕事というものはお客が頼んだことをただそのままやるような人にできるものじゃないと思う。

The customer is always right in the sense that the measure of good design is how well it works for the user. If you make a novel that bores everyone, or a chair that's horribly uncomfortable to sit in, then you've done a bad job, period. It's no defense to say that the novel or the chair is designed according to the most advanced theoretical principles.

もちろん、お客は常に正しい。デザインの良さはユーザに対してもたらす効用によって測られる、という意味においてはね。皆を飽きさせる小説を書いたり、座り心地が最悪な椅子を作ったりしたら、それは単にダメな仕事だ。たとえその小説だか椅子だかが最新の理論に基づいて作られていたとしたって何の足しにもなりやしない。

And yet, making what works for the user doesn't mean simply making what the user tells you to. Users don't know what all the choices are, and are often mistaken about what they really want.

それでも、ユーザにうまく使ってもらえるものを作るというのは、ユーザの言う通りに作るということとは違う。ユーザは全ての選択肢を知っているわけじゃないし、本当に欲しいものが何かということについてはよく間違ふ。

The answer to the paradox, I think, is that you have to design for the user, but you have to design what the user needs, not simply what he says he wants. It's much like being a doctor. You can't just treat a patient's symptoms. When a patient tells you his symptoms, you have to figure out what's actually wrong with him, and treat that.

このパラドックスへの答えが、あなたはユーザのためにデザインしなくちゃならないけれど、単にユーザが欲しいというものではなく、ユーザが必要としているものをデザインしなくちゃならない、ということなんだ。これは医者役割によく似ている。患者の症状だけを診ていてはだめだ。患者が症状を話している時に、何が原因となっているかを見極めて、それを手当てしないと。

This focus on the user is a kind of axiom from which most of the practice of good design can be derived, and around which most design issues center.

こういった、ユーザに注意を向けることは、良いデザインをもたらす方法の多くで、一種の公理のようなものになっている。デザインに関する問題の多くはその周辺にある。

If good design must do what the user needs, who is the user? When I say that design must be for users, I don't mean to imply that good design aims at some kind of lowest common denominator. You can pick any group of users you want. If you're designing a tool, for example, you can design it for anyone from beginners to experts, and what's good design for one group might be bad for another. The point is, you have to pick some group of users. I don't think you can even talk about good or bad design except with reference to some intended user.

良いデザインがユーザの必要としているものを作るんだとして、ではユーザとは誰だろう。私はデザインはユーザのためのものだと言ったが、ユーザの最小公倍数みたいなものを目指すのが良いデザインだというつもりはない。どんなふうにもいいから、あるユーザの集合を決めるんだ。例えば道具をデザインしているなら、初心者向けにデザインしてもいいし、熟練者向けでもいい。ただ、あるグループ向けの良いデザインは、別のグループ向けとしては良くないかもしれない。大事なことは特定のグループのユーザを対象とすることだ。対象とするユーザを特定しないことには、デザインの良否さえ議論できないんじゃないかと思う。

You're most likely to get good design if the intended users include the designer himself. When you design something for a group that doesn't include you, it tends to be for people you consider to be less sophisticated than you, not more sophisticated.

良いデザインが出てきやすいのは、対象とするユーザがあなた自身を含んでいる時だ。自分自身を含まないグループに対して何かをデザインしていると、対象ユーザ層は自分より上ではなく、下だと思ってしまいがちになる。

That's a problem, because looking down on the user, however benevolently, seems inevitably to corrupt the designer. I suspect that very few housing projects in the US were designed by architects who expected to live in them. You can see the same thing in programming languages. C, Lisp, and Smalltalk were created for their own designers to use. Cobol, Ada, and Java, were created for other people to use.

ユーザを見下ろすことは、たとえ慈悲心があったとしても、いずれはデザイナーを駄目にする。アメリカの公営住宅の設計者は、自分でそこに住もうなんて思っていなかったのだろう。同じことはプログラミング言語にも見て取れる。C、Lisp、Smalltalk はデザイナー

が自分で使うために作られた。Cobol、Ada、Java は他人が使うために作られた。

If you think you're designing something for idiots, the odds are that you're not designing something good, even for idiots.

あなたが間抜けのために何かをデザインしているんだとしたら、間抜けにとっても役に立たないものをデザインすることになるのが落ちだろう。

Even if you're designing something for the most sophisticated users, though, you're still designing for humans. It's different in research. In math you don't choose abstractions because they're easy for humans to understand; you choose whichever make the proof shorter. I think this is true for the sciences generally. Scientific ideas are not meant to be ergonomic.

しかし、最も優れたユーザ層のためにデザインしている場合でも、相手にしているのは依然として人間なんだ。そこがリサーチと違うところだ。数学では、人間に分かりやすくするためにある抽象化方法を選ぶなんてことはしない。証明を短くするのに良い抽象化方法を選ぶだけだ。どの科学でも概ねそういうものだとは思う。科学的な思考は使う人にやさしいことを指向してはいない。

Over in the arts, things are very different. Design is all about people. The human body is a strange thing, but when you're designing a chair, that's what you're designing for, and there's no way around it. All the arts have to pander to the interests and limitations of humans. In painting, for example, all other things being equal a painting with people in it will be more interesting than one without. It is not merely an accident of history that the great paintings of the Renaissance are all full of people. If they hadn't been, painting as a medium wouldn't have the prestige that it does.

芸術の分野では事情は全く異なる。デザインは人々のためのものだ。人間の身体というのは奇妙で扱いにくいものだが、椅子をデザインしているならそれを避けて通るわけにはいかない。全ての芸術は人間の興味と限界に迎合しなくちゃならない。例えば絵画では、他の全ての要素が等しければ、人間が描かれているもののほうがより興味深いものになる。ルネッサンスの偉大な絵画が人でいっぱいなのは歴史の偶然ではない。そうでなかったら、媒体としての絵画が今のような地位を得ることはなかっただろう。

Like it or not, programming languages are also for people, and I suspect the

human brain is just as lumpy and idiosyncratic as the human body. Some ideas are easy for people to grasp and some aren't. For example, we seem to have a very limited capacity for dealing with detail. It's this fact that makes programming languages a good idea in the first place; if we could handle the detail, we could just program in machine language.

好むと好まざるとにかかわらず、プログラミング言語も人々のためのものだ。そして人間の頭脳は身体と同じく、でこぼこでへんてこりんなものだと思う。人々にとって理解しやすい考えもあれば、そうでない考えもある。例えば、私たち人間が細部を扱う能力は極めて限られているようだ。だからこそ、プログラミング言語というものが有用になったといえる。私たちが細部を扱うことに障害がなければ、機械語でプログラムしていたってよかったはずだ。

Remember, too, that languages are not primarily a form for finished programs, but something that programs have to be developed in. Anyone in the arts could tell you that you might want different mediums for the two situations. Marble, for example, is a nice, durable medium for finished ideas, but a hopelessly inflexible one for developing new ideas.

また、プログラミング言語は、完成されたプログラムを表現するものというより、プログラムがまさに開発されている最中の形を表現するものだというのを忘れないでほしい。芸術の分野で仕事をしている人なら誰でも、その 2 つの過程には異なる媒体が必要だと言うだろう。例えば、大理石は最終的な考えを保持するには素晴らしく長持ちする媒体だが、新しい考えを発展させている時にはおそろしく融通の利かない媒体でもある。

A program, like a proof, is a pruned version of a tree that in the past has had false starts branching off all over it. So the test of a language is not simply how clean the finished program looks in it, but how clean the path to the finished program was. A design choice that gives you elegant finished programs may not give you an elegant design process. For example, I've written a few macro-defining macros full of nested backquotes that look now like little gems, but writing them took hours of the ugliest trial and error, and frankly, I'm still not entirely sure they're correct.

プログラムというものは、証明のように、もともとさまざまな間違いの枝がそこいらじゅうに生えていた木の枝をきれいに刈り込んだものだ。だから言語の良さを測るには、完成したプログラムがどれだけきれいかを見るだけじゃ駄目で、プログラムが完成へと至った道筋がどれだけきれいだったかを見なければいけない。プログラムの最終形をエレガントにするためのデザイン上の選択は、プログラムそのもののデザインの道筋をきれいにする

ものではないかもしれない。私は入れ子になったバッククオートで固められたマクロ定義を生成するマクロをいくつか書いたことがあり、それは今では小さな宝石のように見えるが、それを書くにあたっては何時間もの醜い試行錯誤を繰り返したし、正直に言って、それが完全に正しいかどうか今でも自信がない。

We often act as if the test of a language were how good finished programs look in it. It seems so convincing when you see the same program written in two languages, and one version is much shorter. When you approach the problem from the direction of the arts, you're less likely to depend on this sort of test. You don't want to end up with a programming language like marble.

私たちはつい、完成したプログラムがどれだけ良く見えるかで言語の良さを測ってしまいがちだ。2つの言語で書かれた同じプログラムを眺めて、片方がもう片方よりうんと短かかったりすると、その議論にはひどく説得力がある。だが芸術の方向からこの問題に取り組んでみれば、そういう測り方をしてしまうことは少ないだろう。プログラミング言語を、大理石のような、完成形は美しいけれどそれを使った製作が苦痛であるような媒体にはしたくはないはずだ。

For example, it is a huge win in developing software to have an interactive toplevel, what in Lisp is called a read-eval-print loop. And when you have one this has real effects on the design of the language. It would not work well for a language where you have to declare variables before using them, for example. When you're just typing expressions into the toplevel, you want to be able to set *x* to some value and then start doing things to *x*. You don't want to have to declare the type of *x* first. You may dispute either of the premises, but if a language has to have a toplevel to be convenient, and mandatory type declarations are incompatible with a toplevel, then no language that makes type declarations mandatory could be convenient to program in.

例を挙げよう。ソフトウェアを開発する時、会話的なトップレベルをつけておくのは非常に有効だ。Lisp では read-eval-print ループと呼ばれているものだ。だが会話的トップレベルを付けるとすれば、それは言語のデザインに確かな影響を与える。例えば変数を前もって宣言しなければならないような言語はうまくないというようなことだ。トップレベルに式を打ち込んでいる時は、まず *x* に何かの値をセットしてそれから *x* をこう使おうという具合に考えているはずだ。そんな時に最初に *x* の型をいちいち宣言したくはない。この前提条件に反論の余地はあるだろうが、もし便利な言語はトップレベルを持たなければならず、そして型宣言を強制することはトップレベルと親和性が悪いということであれば、型宣言を強制する言語は便利な言語には成り得ないということになる。

In practice, to get good design you have to get close, and stay close, to your users. You have to calibrate your ideas on actual users constantly, especially in the beginning. One of the reasons Jane Austen's novels are so good is that she read them out loud to her family. That's why she never sinks into self-indulgently arty descriptions of landscapes, or pretentious philosophizing. (The philosophy's there, but it's woven into the story instead of being pasted onto it like a label.) If you open an average "literary" novel and imagine reading it out loud to your friends as something you'd written, you'll feel all too keenly what an imposition that kind of thing is upon the reader.

実際には、良いデザインを得るにはユーザに近づき、そして常にユーザの側にいなくてはならない。特に最初は実際のユーザに合わせて常に考えを調整しなければならない。特に最初の段階がそうだ。ジェーン・オースティンの小説が良い理由のひとつは、彼女はまず作品を家族に読んで聞かせたからだ。だから彼女は自己満足的な芸術ぶった風景の描写や、もったいぶった哲学的考察に堕ちることが決してなかった。(もちろん彼女の小説に哲学はある。だがそれはお札のようにストーリーの上にぺたぺた貼られているのではなく、ストーリーの中に編み込まれているのだ)。そこらにある「文学」小説を開いて、自分の書いたものの振りをして友人に読み聞かせるところを想像してみたまえ。きっとそれがどれだけ読み手の負担になることか感じられるだろう。

In the software world, this idea is known as Worse is Better. Actually, there are several ideas mixed together in the concept of Worse is Better, which is why people are still arguing about whether worse is actually better or not. But one of the main ideas in that mix is that if you're building something new, you should get a prototype in front of users as soon as possible.

ソフトウェアの世界では、これは「悪いほうが良い」原則として知られている。実は「悪いほうが良い」原則にはいくつかのアイディアが混ざり合っていて、そのせいで人々はいまだに悪いのは本当に良いのかどうか議論を続けているんだが、これらのアイディアの中心となっている考え方は、何か新しいものを作るときはできるだけ早くプロトタイプをユーザの前に出すべきだということだ。

The alternative approach might be called the Hail Mary strategy. Instead of getting a prototype out quickly and gradually refining it, you try to create the complete, finished, product in one long touchdown pass. As far as I know, this is

a recipe for disaster. Countless startups destroyed themselves this way during the Internet bubble. I've never heard of a case where it worked.

もうひとつの方法は聖母マリア戦略と呼ぶのかもしれない。プロトタイプを素早く出して少しずつ改善する代わりに、長い時間をかけて完全な最終形の製品を作り上げようとするものだ。私の知る限り、これは失敗へのタッチダウンパスだ。インターネットバブルの頃には、数え切れないほどのベンチャー企業がこの方法を取って自滅していった。私はこの方法が成功した例を聞いたことがない。

What people outside the software world may not realize is that Worse is Better is found throughout the arts. In drawing, for example, the idea was discovered during the Renaissance. Now almost every drawing teacher will tell you that the right way to get an accurate drawing is not to work your way slowly around the contour of an object, because errors will accumulate and you'll find at the end that the lines don't meet. Instead you should draw a few quick lines in roughly the right place, and then gradually refine this initial sketch.

ソフトウェアの世界の外にいる人々が見落としがちなのは、この「悪いほうが良い」原則は芸術の分野に普遍的に見られるということだ。絵画では、この方法はルネッサンスの時代に発見された。今ではほとんどの絵画の教師が、正確な線画を得るにはものの輪郭を順にゆっくりなぞっていったはいけないと教えるだろう。誤差が積み重なって、最終的に線が合わなくなる。そうではなく、いくつかの線をだいたいの位置にざっと置いてみて、そのスケッチを次第に詳細化してゆくのだ。

In most fields, prototypes have traditionally been made out of different materials. Typefaces to be cut in metal were initially designed with a brush on paper. Statues to be cast in bronze were modelled in wax. Patterns to be embroidered on tapestries were drawn on paper with ink wash. Buildings to be constructed from stone were tested on a smaller scale in wood.

多くの分野では、プロトタイプは伝統的に異なる材料で作られてきた。活字は紙の上で筆書きでデザインされた後に金属に刻まれた。彫像は蝋でモデリングされた後にブロンズに鋳造された。タペストリーに織り込まれる模様はまず紙にインクウォッシュで描かれた。石造りの建物はまず木で小さい模型が造られた。

What made oil paint so exciting, when it first became popular in the fifteenth century, was that you could actually make the finished work *from* the prototype. You could make a preliminary drawing if you wanted to, but you weren't held to it; you could work out all the details, and even make major changes, as you finished

the painting.

油絵が、15 世紀に広まった時に熱狂的に受け入れられたのは、プロトタイプからそのまま最終的な作品を作ることができたからだだった。もちろん下書きを書くことはできるが、そこで止める理由はない。完成するまでどんどん詳細化してゆくことができるし、大きな変更を加えることだってできる。

You can do this in software too. A prototype doesn't have to be just a model; you can refine it into the finished product. I think you should always do this when you can. It lets you take advantage of new insights you have along the way. But perhaps even more important, it's good for morale.

ソフトウェアでも同じことができる。プロトタイプはただの模型に留まる必要はない。それを詳細化して、最終的な製品にまで持っていけるんだ。これは可能ならいつでもそうすべきだと思う。この方法は作ってゆく過程で新しいアイディアを入れていけるし、何より大事なのは、士気を保てることだ。

Morale is key in design. I'm surprised people don't talk more about it. One of my first drawing teachers told me: if you're bored when you're drawing something, the drawing will look boring. For example, suppose you have to draw a building, and you decide to draw each brick individually. You can do this if you want, but if you get bored halfway through and start making the bricks mechanically instead of observing each one, the drawing will look worse than if you had merely suggested the bricks.

士気はデザインの要だ。なんでこのことがもっと話題に上らないのか私は不思議だ。私の最初の絵画の教師はこう教えてくれた。もし何かを描いていて飽きてきたら、画も退屈なものになるだろうと。例えば建物を描いているとしよう。そして煉瓦を一つひとつ描き込もうと思ったとしよう。もちろんやってやれなくはないだろう。しかし、半分まで行って飽きがきて、残りの煉瓦を、一つひとつをよく観察する代わりに機械的に埋めていったとしたら、出来上がりは煉瓦をただ感じさせるだけの仕上げよりもずっと悪いものになるだろう。

Building something by gradually refining a prototype is good for morale because it keeps you engaged. In software, my rule is: always have working code. If you're writing something that you'll be able to test in an hour, then you have the prospect of an immediate reward to motivate you. The same is true in the arts, and

particularly in oil painting. Most painters start with a blurry sketch and gradually refine it. If you work this way, then in principle you never have to end the day with something that actually looks unfinished. Indeed, there is even a saying among painters: "A painting is never finished, you just stop working on it." This idea will be familiar to anyone who has worked on software.

プロトタイプから次第に詳細化してゆく方法が士気に良いのは、常に没頭していただけるからだ。ソフトウェアにおける私のルールはこうだ。常に動くコードを持っている。1 時間後には試せるものを書いていれば、目の前にある見返りによっていつもやる気を保っていただける。芸術でも同じだ。特に油彩画では。多くの画家はぼんやりしたスケッチからはじめて徐々に細部を描き込んで行く。この方法をとれば、原理的に画が未完成に見えるまま一日が終るということがない。実際、画家の間でよく言われるせりふがある。「画には終りが無い。ただ描くのを止めるだけだ」。ソフトウェアの仕事をしたことのある人なら、この考えはよく分かるだろう。

Morale is another reason that it's hard to design something for an unsophisticated user. It's hard to stay interested in something you don't like yourself. To make something good, you have to be thinking, "wow, this is really great," not "what a piece of shit; those fools will love it."

士気の問題は、低いレベルのユーザのために何かをデザインするのが難しいことのもうひとつの理由でもある。自分で気に入らない何かに対して興味を持ち続けるのはとても難しい。良いものを作りたいなら、いつも「うーむ、こいつは素晴らしいぞ」と思い続けていなくちゃ。「なんてひどいがらくただ。馬鹿どもは気に入るだろうがな」なんて考えるんじゃないかね。

Design means making things for humans. But it's not just the user who's human. The designer is human too.

デザインは、人間のために何かを作るということだ。でもユーザだけが人間なんじゃない。デザイナーもまた、人間なのだ。

Notice all this time I've been talking about "the designer." Design usually has to be under the control of a single person to be any good. And yet it seems to be possible for several people to collaborate on a research project. This seems to me one of the most interesting differences between research and design.

ところで、私はずっと「デザイナー」という言葉を単数で使ってきたことに注意してほしい

い。デザインが良いものになるためには、通常はそれが一人の人間の支配下になくなくてはならない。しかし、リサーチのプロジェクトでは何人かの人々が協力し合うことは可能に思える。これは、リサーチとデザインの違いのうちの最も興味深い点だと思う。

There have been famous instances of collaboration in the arts, but most of them seem to have been cases of molecular bonding rather than nuclear fusion. In an opera it's common for one person to write the libretto and another to write the music. And during the Renaissance, journeymen from northern Europe were often employed to do the landscapes in the backgrounds of Italian paintings. But these aren't true collaborations. They're more like examples of Robert Frost's "good fences make good neighbors." You can stick instances of good design together, but within each individual project, one person has to be in control.

芸術の分野でも協力し合った有名な事例がないわけではないが、その多くは核融合みたいなものじゃなく、分子結合のようなものだと思う。オペラでは一人が台本を書いても一人が作曲するのは普通のことだ。絵画でも、画の別々の部分がそれぞれの専門家によって描かれることはあり得る。ルネッサンスの時代、北ヨーロッパから来た職人がしばしばイタリアの絵画の背景を描くために雇われた。だがこれらは本当の共同作業ではない。どちらかというと、フロストが言う「良い柵は良い隣人をつくる」の具体例みたいなものだ。それ自体が良いデザインであるものをくっつけることはできるが、それぞれのプロジェクトの中では、一人の人間がコントロールすることが必要だ。

I'm not saying that good design requires that one person think of everything. There's nothing more valuable than the advice of someone whose judgement you trust. But after the talking is done, the decision about what to do has to rest with one person.

別に、良いデザインをするには一人の人間が全てを考えなくちゃならないと言っているわけじゃない。あなたが信頼できる人の意見ほど貴重なものはない。だが、会話が終わったら、決断は一人の人間に任せなければならない。

Why is it that research can be done by collaborators and design can't? This is an interesting question. I don't know the answer. Perhaps, if design and research converge, the best research is also good design, and in fact can't be done by collaborators. A lot of the most famous scientists seem to have worked alone. But I don't know enough to say whether there is a pattern here. It could be simply that many famous scientists worked when collaboration was less common.

なぜリサーチは協力してできるのにデザインはできないのだろう。これは面白い質問だ。

私には答えはわからない。もしかすると、デザインとリサーチがいずれ合流するとすれば、最高のリサーチは良いデザインでもあるはずで、だとすればそれは協力してはできないものかもしれない。最も有名な科学者たちの多くは一人で仕事をしていたように思える。だがここにパターンがあると言えるほど私はよく知っているわけじゃない。有名な科学者の多くは協同研究がそれほど一般的ではなかった時代にいたと言うだけかもしれない。

Whatever the story is in the sciences, true collaboration seems to be vanishingly rare in the arts. Design by committee is a synonym for bad design. Why is that so? Is there some way to beat this limitation?

科学者の間にどんなストーリーがあろうと、芸術の分野での真の共同作業はほとんど見出せないと言えるだろう。委員会によるデザインは悪いデザインと同義だ。なぜそうなのだろう。この限界を破る方法はあるのだろうか。

I'm inclined to think there isn't-- that good design requires a dictator. One reason is that good design has to be all of a piece. Design is not just for humans, but for individual humans. If a design represents an idea that fits in one person's head, then the idea will fit in the user's head too.

私はその方法はないという考えの方に惹かれる。良いデザインには独裁者が必要なんだ。ひとつの理由は、良いデザインは完全に一つにまとまったものでなければならないというものだろう。デザインは単に人間のためのものであるというだけでなく、個々の人間のためのものだ。デザインが一人の人間の頭に収まる考えを表わしているのなら、それはユーザの頭にもきっと収まるだろう。

Chapter 16 GREAT HACKERS

第 16 章 素晴らしいハッカー

July 2004

(This essay is derived from a talk at Oscon 2004.)

(この記事は OSCON 2004 での基調講演を基にしたものである。)

A few months ago I finished a new [book](#), and in reviews I keep noticing words like "provocative" and "controversial." To say nothing of "idiotic."

私は数ヵ月前に新しい本を出版したんだが、「挑発的」だとか「論議を呼ぶ」だとかいう書評がやけに目立った。「大馬鹿だ」という評については何も言わないでおこう。

I didn't mean to make the book controversial. I was trying to make it efficient. I didn't want to waste people's time telling them things they already knew. It's more efficient just to give them the diffs. But I suppose that's bound to yield an alarming book.

あの本を敢えて議論を呼ぶように書いたつもりはない。むしろ、効率的にしたかったんだ。読者がもう知っていることをまたくどくど説いてみんなの時間を無駄にしたくはなかった。知られてることとの diff だけ示すほうが効率的じゃないか。だが、どうやらこの方法は危ない本を産み出してしまうようだ。

Edisons

エジソン

There's no controversy about which idea is most controversial: the suggestion that variation in wealth might not be as big a problem as we think.

どれが最も論議を呼ぶアイデアかという点に関しては、さして論議がなかったようだ。富の格差は広く思われているほどには大きな問題ではない、と言ったやつだ。

I didn't say in the book that variation in wealth was in itself a good thing. I said in some situations it might be a sign of good things. A throbbing headache is not a good thing, but it can be a sign of a good thing-- for example, that you're recovering consciousness after being hit on the head.

私は本の中で、富の格差がそれ自体として良いものだと言ったわけじゃない。私が言ったのは、ある状況下では、富の格差は良いものを示すサインだということだ。頭がずきんず

きんと痛むというのは、それ自体は良いことではないが、良いことを示すサインではあり得る。例えば、頭を強く打った後で、意識を取り戻しつつある場合なんかだ。

Variation in wealth can be a sign of variation in productivity. (In a society of one, they're identical.) And *that* is almost certainly a good thing: if your society has no variation in productivity, it's probably not because everyone is Thomas Edison. It's probably because you have no Thomas Edisons.

富の格差は、生産性の格差を示している可能性がある(一人だけからなる社会では、両者は等しい)。だとしたら、それ自身はほぼ確実に良いことだと言える。生産性に格差がない社会があったとしたら、それは皆がトーマス・エジソンである社会であるよりは、トーマス・エジソンがいない社会である可能性の方が高い。

In a low-tech society you don't see much variation in productivity. If you have a tribe of nomads collecting sticks for a fire, how much more productive is the best stick gatherer going to be than the worst? A factor of two? Whereas when you hand people a complex tool like a computer, the variation in what they can do with it is enormous.

技術が低い社会では生産性の格差はそれほど見られない。遊牧民の一族が焚火のために枝を集めているとしよう。一番薪集めがうまい人と一番下手な人の差はどれくらいのものだろう? せいぜい2倍くらいだろうか。だが、人々がコンピュータのような複雑な道具を手にしたなら、それでできることの差は極めて大きいものになる。

That's not a new idea. Fred Brooks wrote about it in 1974, and the study he quoted was published in 1968. But I think he underestimated the variation between programmers. He wrote about productivity in lines of code: the best programmers can solve a given problem in a tenth the time. But what if the problem isn't given? In programming, as in many fields, the hard part isn't solving problems, but deciding what problems to solve. Imagination is hard to measure, but in practice it dominates the kind of productivity that's measured in lines of code.

これは新しいアイデアではない。フレッド・ブルックスはこのことを 1974 年に書いているし、そこで彼が引用している研究は 1968 年に発表されたものだ。ただ、彼はプログラマ間の差を低く見積っていると私は思う。彼は生産性を、コードの行数で測って、一番のプログラマは与えられた問題を 10 分の 1 の時間で解くとした。でも、問題が与えられていなかったとしたら? 他の多くの分野でも同じだが、プログラミングでも、難しいのは問題を解くことではなく、どの問題を解くかを決めることだ。想像力を測るのは難しい。が、

現実にはそれはコードの行数で測られるような生産性をはるかに凌駕するだろう。

Productivity varies in any field, but there are few in which it varies so much. The variation between programmers is so great that it becomes a difference in kind. I don't think this is something intrinsic to programming, though. In every field, technology magnifies differences in productivity. I think what's happening in programming is just that we have a lot of technological leverage. But in every field the lever is getting longer, so the variation we see is something that more and more fields will see as time goes on. And the success of companies, and countries, will depend increasingly on how they deal with it.

どんな分野でも生産性の差はあるが、それが非常に大きい分野というのは限られる。プログラマ間の差はあまりに大きく、そのため上のほうと下のほうでは別の種類のプログラマと言ってもよいくらいだ。だが、これはプログラミングに固有のものではないと思う。どの分野でも、技術は生産性の差を拡大する。プログラミングの世界で起こっているのは、単に技術による槌子の効果が非常に大きく効いているってことだ。あらゆる分野でこの槌子は次第に長くなっており、したがって時が経つにつれ、より多くの分野でこのような差が見られるようになってゆくだろう。そして、会社や国の成功も、この効果をどう扱うかにより大きく依存するようになってゆくだろう。

If variation in productivity increases with technology, then the contribution of the most productive individuals will not only be disproportionately large, but will actually grow with time. When you reach the point where 90% of a group's output is created by 1% of its members, you lose big if something (whether Viking raids, or central planning) drags their productivity down to the average.

技術に伴って生産性の差が開いていくとしたら、最も生産性の高い個人の貢献は、それが全体の中で不釣り合いに大きなだけでなく、さらに大きくなってゆくだろう。集団の出力の90%が、1%のメンバーによって作られる点にまで達したら、彼らの生産性を何か(海賊の襲撃でも、中央集権管理でも)によって平均まで引きずり降ろすことは、巨大な損失を意味することになる。

If we want to get the most out of them, we need to understand these especially productive people. What motivates them? What do they need to do their jobs? How do you recognize them? How do you get them to come and work for you? And then of course there's the question, how do you become one?

これらの特に生産的な人々から最も多くを得るには、彼らを理解する必要がある。彼らはどういう動機を持っているんだろう。仕事をするのに必要なものは何だろう。そういう人々

をどうやって見つけられるだろう。そういう人々に仕事をしてもらうにはどうすればいいだろう。そしてもちろん、これも大事な質問だ。どうしたらそういう人になれるだろう？

More than Money

お金よりも

I know a handful of super-hackers, so I sat down and thought about what they have in common. Their defining quality is probably that they really love to program. Ordinary programmers write code to pay the bills. Great hackers think of it as something they do for fun, and which they're delighted to find people will pay them for.

私は数人のスーパーハッカーを知っているので、彼らが共通に持っているものは何かを考えてみた。彼らの特徴づけている資質は、おそらく、プログラミングを本当に愛しているということだ。普通のプログラマは生活のためにコードを書く。素晴らしいハッカーにとっては、コード書きは楽しみのためにするもので、それに金を払ってくれるという人がいれば大いに喜ぶのだ。

Great programmers are sometimes said to be indifferent to money. This isn't quite true. It is true that all they really care about is doing interesting work. But if you make enough money, you get to work on whatever you want, and for that reason hackers *are* attracted by the idea of making really large amounts of money. But as long as they still have to show up for work every day, they care more about what they do there than how much they get paid for it.

偉大なプログラマは金に関心がない、と言われることもある。これは必ずしも正確ではない。彼らが本当に関心があるのは確かに面白い仕事をするということだ。そして、十分に金を儲ければ、やりたい仕事がやれる。この理由から、ハッカーは大きな財をなすことに惹かれるんだ。でも、どうせ毎日出勤しなければならないのだとしたら、どれだけ給与をもらえるかよりは、そこで何をするかのほうがハッカーにとっては大事なことだ。

Economically, this is a fact of the greatest importance, because it means you don't have to pay great hackers anything like what they're worth. A great programmer might be ten or a hundred times as productive as an ordinary one, but he'll consider himself lucky to get paid three times as much. As I'll explain later, this is partly because great hackers don't know how good they are. But it's also because money is not the main thing they want.

経済的には、この事実は重要な意味を持つ。素晴らしいハッカーに、彼の価値相応のもの

を支払う必要がないということだからだ。凄腕のプログラマは普通のプログラマの **10** 倍から **100** 倍生産的かもしれないが、普通のプログラマの **3** 倍も給料をもらえばラッキーだと思ってくれるだろう。後で説明するように、すごいハッカーは自分がどれだけすごいのか自分では分からない、ということも、その理由のひとつだ。だが、彼らの本当に欲しいものが金ではない、ということもまたその理由だろう。

What do hackers want? Like all craftsmen, hackers like good tools. In fact, that's an understatement. Good hackers find it unbearable to use bad tools. They'll simply refuse to work on projects with the wrong infrastructure.

ハッカーが欲しがるものは何だろう?他のものづくりの人々と同じように、ハッカーは良い道具が好きだ。いや、これは控えめに過ぎるかもしれない。良いハッカーは、悪い道具を使うことに耐えられない。彼らは、間違ったインフラストラクチャを使うプロジェクトで働くことを、単に拒絶するだろう。

At a startup I once worked for, one of the things pinned up on our bulletin board was an ad from IBM. It was a picture of an AS400, and the headline read, I think, "hackers despise it." [\[1\]](#)

私が以前働いていたベンチャーでは、掲示板に **IBM** の広告がピンで止めてあった。それは **AS400** の写真で、その上に付けられていた表題は確か、「ハッカーはこれを嫌悪する」だったと思う。 [\[1\]](#)

When you decide what infrastructure to use for a project, you're not just making a technical decision. You're also making a social decision, and this may be the more important of the two. For example, if your company wants to write some software, it might seem a prudent choice to write it in Java. But when you choose a language, you're also choosing a community. The programmers you'll be able to hire to work on a Java project won't be as [smart](#) as the ones you could get to work on a project written in Python. And the quality of your hackers probably matters more than the language you choose. Though, frankly, the fact that good hackers prefer Python to Java should tell you something about the relative merits of those languages.

プロジェクトに使うインフラを決定するということは、単なる技術的な決定ではない。それは社会的な決定でもあり、両者のうちではむしろ後者のほうが重要かもしれない。例えば、あなたの会社がソフトウェアを書こうとした時、**Java** を選ぶのが、分別ある態度に見えるかもしれない。しかし、言語を選ぶというのは、同時にコミュニティを選んでいることにもなる。**Java** のプロジェクトで働くために雇われるプログラマは、**Python** を使うプ

プロジェクトで雇えるプログラマほど賢くはないだろう。そして、雇えるハッカーの質は、たぶん言語の選択よりもずっと重要だ。もっとも、正直に言えば、良いハッカーは **Java** より **Python** を好むという事実が、これらの言語の相対的な利点について何かを暗示していると思う。

Business types prefer the most popular languages because they view languages as standards. They don't want to bet the company on Betamax. The thing about languages, though, is that they're not just standards. If you have to move bits over a network, by all means use TCP/IP. But a programming language isn't just a format. A programming language is a medium of expression.

ビジネスの連中は、最も人気のある言語を好む。言語は標準みたいなものだと思っているからだ。Betamax に社運を賭けたくはないだろうからね。でも、言語に関する問題は、それが単なる標準ではないということだ。今、ネットワーク上でビットをやりとりするなら、何はさておき TCP/IP を使うべきだろう。だが、プログラミング言語は単なるフォーマットではない。プログラミング言語は表現の媒体なんだ。

I've read that Java has just overtaken Cobol as the most popular language. As a standard, you couldn't wish for more. But as a medium of expression, you could do a lot better. Of all the great programmers I can think of, I know of only one who would voluntarily program in Java. And of all the great programmers I can think of who don't work for Sun, on Java, I know of zero.

Java は最も人気のある言語として Cobol を追い抜いた、という記事を読んだ。標準としては、それ以上望むことはないだろう。だが、表現手段としては、まだまだ改善の余地があるはずだ。私が思い付く全ての偉大なプログラマのうち、進んで **Java** を書いている人物は一人しかいない。そして私が思い付く全ての偉大なプログラマで、**Sun** で働いていなくて、**Java** で書いている人物というのは、一人もいない。

Great hackers also generally insist on using open source software. Not just because it's better, but because it gives them more control. Good hackers insist on control. This is part of what makes them good hackers: when something's broken, they need to fix it. You want them to feel this way about the software they're writing for you. You shouldn't be surprised when they feel the same way about the operating system.

素晴らしいハッカーはまた、一般的に、オープンソースソフトウェアを使うことを主張する。そちらのほうが良質だというだけでなく、自分で多くをコントロールできるからだ。良いハッカーはコントロールしたがる。それが彼らを良いハッカーたらしめている理由の

ひとつだ。何かが壊れたら、自分で直したいんだ。あなたがハッカーにプログラムを書いてもらっているとすれば、もちろん、そのプログラムが壊れたらハッカー自身に直してもらいたいだろう。でも、ハッカーはオペレーティングシステムにさえ、全く同様に感じていると聞いても驚いてはいけない。

A couple years ago a venture capitalist friend told me about a new startup he was involved with. It sounded promising. But the next time I talked to him, he said they'd decided to build their software on Windows NT, and had just hired a very experienced NT developer to be their chief technical officer. When I heard this, I thought, these guys are doomed. One, the CTO couldn't be a first rate hacker, because to become an eminent NT developer he would have had to use NT voluntarily, multiple times, and I couldn't imagine a great hacker doing that; and two, even if he was good, he'd have a hard time hiring anyone good to work for him if the project had to be built on NT. [2]

2年ほど前、友人のベンチャーキャピタリストが、彼が関わっている新しいベンチャーについて話してくれた。その時は、それはかなり有望に思えた。次に彼と会った時、そのベンチャーはソフトウェアを Windows NT 上で開発することを決定し、経験豊かな NT 開発者を技術最高責任者として招いたと彼は話した。これを聞いて私は、ああそのベンチャーの運命は決まったと思ったものだ。第一に、その技術最高責任者は一級のハッカーではあり得ない。だって素晴らしい NT 開発者になるためには、NT を何度も何度も、自分から進んで使わなくちゃならないだろう。偉大なハッカーがそんなことをするなんて想像もできない。第二に、もし彼が優秀だったとしても、プロジェクトが NT 上に構築されるなら、優秀な人物を集めることが困難だろうからだ。[2]

The Final Frontier

最後の開拓地

After software, the most important tool to a hacker is probably his office. Big companies think the function of office space is to express rank. But hackers use their offices for more than that: they use their office as a place to think in. And if you're a technology company, their thoughts are your product. So making hackers work in a noisy, distracting environment is like having a paint factory where the air is full of soot.

ハッカーにとって、ソフトウェアの次に大事な道具はおそらく仕事場だ。大企業は、オフィス空間の機能は地位を示すことだと考えている。だがハッカーにとって、仕事場はそれ以上のものだ。オフィスは、その中で考えるためのものなんだ。技術を売る会社にとって、

ハッカーの考えることは会社の製品そのものだと言える。ハッカーを、ざわついて気が散る環境で働かせるのは、ペンキ工場を煤だらけにしておくようなものだ。

The cartoon strip Dilbert has a lot to say about cubicles, and with good reason. All the hackers I know despise them. The mere prospect of being interrupted is enough to prevent hackers from working on hard problems. If you want to get real work done in an office with cubicles, you have two options: work at home, or come in early or late or on a weekend, when no one else is there. Don't companies realize this is a sign that something is broken? An office environment is supposed to be something that *helps* you work, not something you work despite.

ディルバートにはオフィスのブースについてたくさんのエピソードがあるが、それにはもっともな理由があるわけだ。私の知っているハッカーは皆ブースを嫌っている。考えを中断されるかもしれないという可能性だけで、難しい問題を考えようという気が起こらなくなる。仕切りだけのオフィスでまともな仕事をしようと思うなら、選択肢は2つしかない。家で働くか、早朝、深夜、週末といった時間外で誰もいない時に出社するかだ。会社は、プログラマーがそういうことをしだしたら、それが何かがおかしいサインだと気付かないんだろうか。オフィス環境というのは、その中で仕事をするためのものであって、そういう環境にもかかわらず仕事をする、というものじゃないはずだ。

Companies like Cisco are proud that everyone there has a cubicle, even the CEO. But they're not so advanced as they think; obviously they still view office space as a badge of rank. Note too that Cisco is famous for doing very little product development in house. They get new technology by buying the startups that created it-- where presumably the hackers did have somewhere quiet to work.

Cisco のような会社は、全員が、最高経営責任者も含めて、仕切りだけのブースで仕事をしていることを誇りにしている。だが、そういう会社は彼らが自分で考えているほど進んでいるわけじゃない。オフィス空間は階級バッジみたいなものだという考えから抜け出せてないからだ。Cisco は社内ではほとんど開発を行っていないことで有名だということも、思い出してほしい。新しい技術は、それをつくり出したベンチャーを買収することで手に入れている。そういうベンチャーでは、たぶんハッカーは静かな労働環境を持っていたらろう。

One big company that understands what hackers need is Microsoft. I once saw a recruiting ad for Microsoft with a big picture of a door. Work for us, the premise was, and we'll give you a place to work where you can actually get work done. And you know, Microsoft is remarkable among big companies in that they are able to develop software in house. Not well, perhaps, but well enough.

ハッカーが欲するものを理解している大企業がひとつある。マイクロソフトだ。大きなドアが書かれた、マイクロソフトのリクルート広告を見たことがある。それはこう言っていた。「ここに来て働きませんか。私たちは、仕事がちゃんとできる環境を用意しています」。実際、マイクロソフトは、大企業でありながら社内でソフトウェアが開発できているという点で非常に特殊だ。素晴らしくはないかもしれないが、十分なだけのソフトは作り出せている。

If companies want hackers to be productive, they should look at what they do at home. At home, hackers can arrange things themselves so they can get the most done. And when they work at home, hackers don't work in noisy, open spaces; they work in rooms with doors. They work in cosy, neighborhoody places with people around and somewhere to walk when they need to mull something over, instead of in glass boxes set in acres of parking lots. They have a sofa they can take a nap on when they feel tired, instead of sitting in a coma at their desk, pretending to work. There's no crew of people with vacuum cleaners that roars through every evening during the prime hacking hours. There are no meetings or, God forbid, corporate retreats or team-building exercises. And when you look at what they're doing on that computer, you'll find it reinforces what I said earlier about tools. They may have to use Java and Windows at work, but at home, where they can choose for themselves, you're more likely to find them using Perl and Linux.

会社がハッカーに生産的になってほしいと思うなら、ハッカーが家でどうしているかを見るべきだ。家では、ハッカーは仕事はかどるように自分でレイアウトを決められる。ハッカーは家で働く時、騒がしいあけっぴろげの空間では仕事をしていないだろう。ドア付きの部屋で仕事をしているはずだ。広大な駐車場の真ん中のガラス箱みたいなところじゃなく、こじんまりして人が近くにいて、何かを考えたい時はちょっと歩き回ることができるような場所で仕事をしているはずだ。疲れた時は机に向かって働いてるふりをして気を失うんじゃなく、ちょっと昼寝ができるソファがあったりしてね。一番ハックがのってきた夕方にうるさい掃除機を使う掃除スタッフもいない。ミーティングだの、会社の合宿だの(うげえ)、チームビルディングエクササイズだのはもちろんない。そういう環境にいるハッカーの画面を見てみれば、私が上で道具について言ったことが確認できるだろう。そういうハッカーは会社では **Java** と **Windows** を使っているかもしれないが、家では、自分が選べる環境では、**Perl** と **Linux** を使っていることが多いはずだ。

Indeed, these statistics about Cobol or Java being the most popular language can be misleading. What we ought to look at, if we want to know what tools are best,

is what hackers choose when they can choose freely-- that is, in projects of their own. When you ask that question, you find that open source operating systems already have a dominant market share, and the number one language is probably Perl.

実際、Cobol や Java が最も人気ある言語だというような統計は誤解のもとでさえある。何が最高の道具かを知りたいなら、我々が見なければならないのは、ハッカーが自由に選べる際に、つまり、彼ら自身のプロジェクトに、何を選ぶかということだ。この質問を試してみれば、既にオープンソースの OS が圧倒的なシェアを占めており、一番の言語はたぶん Perl だということに気付くだろう。

Interesting

面白いこと

Along with good tools, hackers want interesting projects. What makes a project interesting? Well, obviously overtly sexy applications like stealth planes or special effects software would be interesting to work on. But any application can be interesting if it poses novel technical challenges. So it's hard to predict which problems hackers will like, because some become interesting only when the people working on them discover a new kind of solution. Before ITA (who wrote the software inside Orbitz), the people working on airline fare searches probably thought it was one of the most boring applications imaginable. But ITA made it interesting by [redefining](#) the problem in a more ambitious way.

ハッカーは、良い道具とともに、面白いプロジェクトを欲する。何がプロジェクトを面白くするだろう? もちろん、飛び抜けてセクシーなアプリケーション、ステルス飛行機だとか特殊効果のソフトウェアだとかは、やって面白いプロジェクトだろう。でも、新しい技術的な挑戦があれば、どんなアプリケーションだって面白くなり得る。だから、ハッカーがどういう問題を好むかを予想することは難しい。その問題に取り組んでみて、新しい種類の解法を見つけ出してはじめて、面白くなる、というものもあるからだ。Orbitz で使われているソフトウェアを ITA が書くまでは、航空機の料金検索ソフトウェアを書いている人々は、たぶん考え得る限りで一番退屈なソフトを書いていると思っていたんじゃないだろうか。だが ITA は問題をずっと野心的な方法で再定義することで、それを面白いものにしたんだ。

I think the same thing happened at Google. When Google was founded, the conventional wisdom among the so-called portals was that search was boring and unimportant. But the guys at Google didn't think search was boring, and that's

why they do it so well.

同じことが **Google** でも起こったんだと思う。**Google** が創立された時、いわゆるポータルについては、検索という部分は退屈でさして重要でないものだというのが一般的な認識だった。だが **Google** は検索は退屈なものとは思わなかった。それが、彼らが検索を非常にうまくやってのけた理由だ。

This is an area where managers can make a difference. Like a parent saying to a child, I bet you can't clean up your whole room in ten minutes, a good manager can sometimes redefine a problem as a more interesting one. Steve Jobs seems to be particularly good at this, in part simply by having high standards. There were a lot of small, inexpensive computers before the Mac. He redefined the problem as: make one that's beautiful. And that probably drove the developers harder than any carrot or stick could.

ここは、マネージャで差をつけられる領域だ。親が子供に言い聞かせるように、部屋を 10 分で綺麗に片付けることはできないかもしれないけれど、良いマネージャは問題をより面白いものに再定義してやることができる。スティーブ・ジョブスはこの点が特に長けている。高いスタンダードを持つというのもその一部だ。**Mac** の前にも、たくさんの、小さく安いコンピュータがあった。彼は問題をこう再定義したんだ——美しいコンピュータを作ろう。それこそが、他のどんな鼻先のエンジンよりも、開発者たちを駆り立てたんじゃないだろうか。

They certainly delivered. When the Mac first appeared, you didn't even have to turn it on to know it would be good; you could tell from the case. A few weeks ago I was walking along the street in Cambridge, and in someone's trash I saw what appeared to be a Mac carrying case. I looked inside, and there was a Mac SE. I carried it home and plugged it in, and it booted. The happy Macintosh face, and then the finder. My God, it was so simple. It was just like ... Google.

そして、彼らは確かにそれに成功した。**Mac** が初めて現れた時、あなたはそれが良いものだを知るのに、電源を入れてみる必要さえなかったはずだ。ケースを見るだけで分かった。何週間か前に、私はケンブリッジの道端を歩いていて、誰かの捨てたごみの中に **Mac** のキャリングケースみたいなものを見つけた。中を覗いて見たら、果たせるかな、**Mac SE** が鎮座していた。私はそれを抱えて帰って、電源を入れてブートしてみた。**Macintosh** の笑った顔が現れて、ファインダーが現れた。ああ、なんてシンプルなんだ。これはまさに…… **Google** のようだ。

Hackers like to work for people with high standards. But it's not enough just to be

exacting. You have to insist on the right things. Which usually means that you have to be a hacker yourself. I've seen occasional articles about how to manage programmers. Really there should be two articles: one about what to do if you are yourself a programmer, and one about what to do if you're not. And the second could probably be condensed into two words: give up.

ハッカーは高い基準を持っている人々と働きたがる。それも、ただ正確にやるだけじゃ十分じゃない。正しいことを主張しなければならない。このことは通常、ハッカーと働く人自身もハッカーでなければならないということを意味する。プログラマをどう管理すべきかを論じた記事をしばしば目にしてきた。実際は、そういう記事は 2 つしか要らないはずだ。ひとつは、あなた自身がプログラマである場合にハッカーたちをどう扱うかというもの。そしてもうひとつは、あなたがそうではない場合のもの。そして、2 番目の記事は一言に要約できるだろう。「あきらめろ」。

The problem is not so much the day to day management. Really good hackers are practically self-managing. The problem is, if you're not a hacker, you can't tell who the good hackers are. A similar problem explains why American cars are so ugly. I call it the *design paradox*. You might think that you could make your products beautiful just by hiring a great designer to design them. But if you yourself don't have good [taste](#), how are you going to recognize a good designer? By definition you can't tell from his portfolio. And you can't go by the awards he's won or the jobs he's had, because in design, as in most fields, those tend to be driven by fashion and schmoozing, with actual ability a distant third. There's no way around it: you can't manage a process intended to produce beautiful things without knowing what beautiful is. American cars are ugly because American car companies are run by people with bad taste.

毎日の管理はさほど問題ではない。本当に優秀なハッカーは自己管理がちゃんとできるからだ。本当の問題は、もしあなたがハッカーでなかったとしたら、誰が良いハッカーかを知ることができないということだ。似たような問題は、米国の車が非常に醜いことの理由にもある。私はこれをデザインのパラドックスと呼んでいる。素晴らしいデザイナーを雇ってデザインさせれば、素晴らしい製品ができると思うかもしれない。でも、自分自身が良いセンスを持っていなかったら、どうやって素晴らしいデザイナーを見つけられる? 定義により、デザイナーの作品をあなたが見ても分からないよね。デザイナーの取った賞だとかこれまでの経歴を見てもあまり役に立たない。デザインの世界では、他の多くの分野でも同じだが、賞や地位というものは流行と評判によるものが非常に大きく、実際の能力ははるかに離れた 3 番目の影響力しか持たないからだ。この問題を避ける方法はない。美とは何かを知らずして、美しいものをつくり出すプロセスを管理することはできないんだ。

米国の車が醜いのは、米国の車会社がセンスのない人々によって運営されているからだ。

Many people in this country think of taste as something elusive, or even frivolous. It is neither. To drive design, a manager must be the most demanding user of a company's products. And if you have really good taste, you can, as Steve Jobs does, make satisfying you the kind of problem that good people like to work on.

この国の多くの人々は、センスというものを何かつかみどころのないものか、取るに足らないものと考えている。どちらも間違いだ。デザインを進めるには、マネージャは会社の製品について最もうるさいユーザでなければならない。そしてあなたが、スティーブ・ジョブスのように本当に良いセンスを持っているなら、優秀な人々が働きたがるような問題を見つけることで自分自身を満足させることができるだろう。

Nasty Little Problems

嫌な細かい問題ども

It's pretty easy to say what kinds of problems are not interesting: those where instead of solving a few big, clear, problems, you have to solve a lot of nasty little ones. One of the worst kinds of projects is writing an interface to a piece of software that's full of bugs. Another is when you have to customize something for an individual client's complex and ill-defined needs. To hackers these kinds of projects are the death of a thousand cuts.

どんな問題がつまらないかを言うのはとても簡単だ。少数の大きく明確に定義された問題を解く代わりに、細かくいやらしい問題をたくさん解かねばならないようなものだ。最悪のプロジェクトのひとつは、バグだらけのソフトウェアのインタフェースを書くことだ。あるいはまた、それぞれの顧客の複雑で定義が曖昧な要求に応じてカスタマイズを行うようなものもそうだ。ハッカーにとって、この手のプロジェクトは少しずつ切り刻まれながら死に至るようなものだ。

The distinguishing feature of nasty little problems is that you don't learn anything from them. Writing a compiler is interesting because it teaches you what a compiler is. But writing an interface to a buggy piece of software doesn't teach you anything, because the bugs are random. [\[3\]](#) So it's not just fastidiousness that makes good hackers avoid nasty little problems. It's more a question of self-preservation. Working on nasty little problems makes you stupid. Good hackers avoid it for the same reason models avoid cheeseburgers.

こういう嫌な細かい問題の何が嫌かということ、それから学べることが何もないということ

だ。コンパイラを書くのは面白い。それによって、コンパイラとは何かが学べるからだ。バグだらけのソフトのインタフェースを書くことからは何も学べない。バグはランダムだからだ。[3] ハッカーは単に気難しいから嫌な細かい問題を避けるんじゃない。それは、むしろ自己防衛の問題なんだ。嫌な細かい問題に関わっていると、馬鹿になる。良いハッカーがそれを避けるのは、ファッションモデルがチーズバーガーを避けるのと同じ理由だ。

Of course some problems inherently have this character. And because of supply and demand, they pay especially well. So a company that found a way to get great hackers to work on tedious problems would be very successful. How would you do it?

いくつかの問題は本質的にそういう性質を持っている。そして需要と供給の関係から、そういう問題に対しては高い報酬が支払われる。だから、偉大なハッカーをひどい問題に取り組ませる方法を見つけた会社は大成功するだろう。どうしたらそれが可能だろうか。

One place this happens is in startups. At our startup we had Robert Morris working as a system administrator. That's like having the Rolling Stones play at a bar mitzvah. You can't hire that kind of talent. But people will do any amount of drudgery for companies of which they're the founders. [4]

そういう状況が起きる場所のひとつはベンチャー企業だ。私たちが立ち上げたベンチャーでは、ロバート・モリスがシステム管理者をやっていた。これはまるで、バル・ミツバーのバックバンドにローリングストーンズを呼ぶみたいなものだ。普通はそんなことのために雇える才能ではない。ただ、人は自分が設立した会社のためなら、どんな苦労も惜しまないものだ。[4]

Bigger companies solve the problem by partitioning the company. They get smart people to work for them by establishing a separate R&D department where employees don't have to work directly on customers' nasty little problems. [5] In this model, the research department functions like a mine. They produce new ideas; maybe the rest of the company will be able to use them.

大企業では、会社を分割することでこの問題を解いている。分離した研究開発部門を作って優秀な人々をそこに集め、彼らが顧客の嫌な細かい問題に直接煩わされずに済むようにする。[5] このモデルでは、研究部門は鉱山のようなものだ。そこは新しいアイデアを産出する。会社の残りの部分は、それを使って何かできるかもしれない。

You may not have to go to this extreme. [Bottom-up programming](#) suggests another way to partition the company: have the smart people work as toolmakers.

If your company makes software to do x, have one group that builds tools for writing software of that type, and another that uses these tools to write the applications. This way you might be able to get smart people to write 99% of your code, but still keep them almost as insulated from users as they would be in a traditional research department. The toolmakers would have users, but they'd only be the company's own developers. [6]

ここまで極端にやらなくてもよいかもしれない。ボトムアッププログラミングは会社を分割する別の方法に使える。優秀な人々に道具を作らせるんだ。会社が x をするソフトウェアを書くとすると、まずひとつのグループにその種のソフトウェアを書くのに適した道具を作らせ、別のグループにそれを使ってアプリケーションを作らせるんだ。こうすれば、優秀な人々が 99% のコードを書きつつ、伝統的な研究部門と同じように彼らを守っておくことができる。道具の製作者にもユーザーがいるが、それはその会社の開発者たちだけ、ということになる。[6]

If Microsoft used this approach, their software wouldn't be so full of security holes, because the less smart people writing the actual applications wouldn't be doing low-level stuff like allocating memory. Instead of writing Word directly in C, they'd be plugging together big Lego blocks of Word-language. (Duplo, I believe, is the technical term.)

マイクロソフトがこの方法をとれば、彼らのソフトウェアはこんなにセキュリティホールだらけにはならないだろう。実際のアプリケーションを開発している、特別に優秀でない人々は、メモリをアロケートするといった低レベルの階層に触らなくて済むだろうからだ。Word を C で直接書くかわりに、彼らは Word 記述言語をレゴブロックのように組み合わせでゆくことになるだろう(技術用語では、Duplo と呼ばれているものだと思う)。

Clumping

ハッカーの集まり

Along with interesting problems, what good hackers like is other good hackers. Great hackers tend to clump together-- sometimes spectacularly so, as at Xerox Parc. So you won't attract good hackers in linear proportion to how good an environment you create for them. The tendency to clump means it's more like the square of the environment. So it's winner take all. At any given time, there are only about ten or twenty places where hackers most want to work, and if you aren't one of them, you won't just have fewer great hackers, you'll have zero. 興味深い問題の他に、良いハッカーが好むのは、他の良いハッカーだ。素晴らしいハッカ

一は寄り集まる傾向がある。ときには信じられないほど——例えば Xerox Parc のように。だから、ハッカーにとって良い環境を作ったとしても、その良さに比例して良いハッカーが集まって来るわけではない。寄り集まる傾向のために、むしろ環境の良さの 2 乗に比例するだろう。これはすなわち、勝者が総取りしてしまうということだ。どの時点でも、おそらくハッカーがぜひ働きたいと思うような場所はせいぜい 10 か 20 くらいしかない。それ以外の場所では、素晴らしいハッカーは少ないというのではなく、全くいないんだ。

Having great hackers is not, by itself, enough to make a company successful. It works well for Google and ITA, which are two of the hot spots right now, but it didn't help Thinking Machines or Xerox. Sun had a good run for a while, but their business model is a down elevator. In that situation, even the best hackers can't save you.

素晴らしいハッカーを雇っておくことだけでは、会社の成功には十分ではない。Google と ITA では——これは現在のホットスポットのうちの 2 つだが——うまくいったが、Thinking Machines 社や Xerox ではうまくいかなかった。Sun はしばらくの間うまくやっていたが、彼らのビジネスモデルはいまや下降するエレベータだ。そのような状況では、最も素晴らしいハッカーであっても会社を救うことはできないだろう。

I think, though, that all other things being equal, a company that can attract great hackers will have a huge advantage. There are people who would disagree with this. When we were making the rounds of venture capital firms in the 1990s, several told us that software companies didn't win by writing great software, but through brand, and dominating channels, and doing the right deals.

けれども、他の条件が全て同じなら、偉大なハッカーを引き付ける会社には巨大なアドバンテージがある。これに同意しない人々もいるだろう。私たちが 1990 年代にベンチャーキャピタル企業から資金集めをしていた時に、いくつかのベンチャーキャピタルはこう言ったものだ。ソフトウェア会社はすごいソフトを書くことで勝つんじゃない、ブランドと独占チャネルと、うまい取り引きで勝つんだ、とね。

They really seemed to believe this, and I think I know why. I think what a lot of VCs are looking for, at least unconsciously, is the next Microsoft. And of course if Microsoft is your model, you shouldn't be looking for companies that hope to win by writing great software. But VCs are mistaken to look for the next Microsoft, because no startup can be the next Microsoft unless some other company is prepared to bend over at just the right moment and be the next IBM.

彼らは本当にそれを信じているようだった。思うに、私にはその理由が分かる。ベンチャ

一キャピタルの多くが、少なくとも無意識のうちに求めているのは、次のマイクロソフトなのだろう。マイクロソフトがモデルなら、もちろんすごいソフトを書いて勝とうなんて考えている会社を探しちゃだめだ。でも、次のマイクロソフトを探すベンチャーキャピタルは間違っているんだ。あるベンチャーが次のマイクロソフトになるためには、ちょうどいい時期に沈んで、次の **IBM** になってくれる会社が不可欠だからだ。

It's a mistake to use Microsoft as a model, because their whole culture derives from that one lucky break. Microsoft is a bad data point. If you throw them out, you find that good products do tend to win in the market. What VCs should be looking for is the next Apple, or the next Google.

マイクロソフトの文化は、その幸運なチャンスをものにしたところから全て始まっているから、それをモデルにするのは間違っている。マイクロソフトはデータ上の特異点なんだ。それを捨ててしまえば、良い製品を作っているところがマーケットで勝っているのを確かに見ることができるだろう。ベンチャーキャピタルが探すべきなのは、次のアップルであり、次の **Google** であるべきなんだ。

I think Bill Gates knows this. What worries him about Google is not the power of their brand, but the fact that they have better hackers. [\[7\]](#)

ビル・ゲイツはこのことを知っていると思う。彼が **Google** を恐れるのは、彼らのブランドではない。Google のほうに良いハッカーがいるという事実だ。 [\[7\]](#)

Recognition

良いハッカーを見分ける

So who are the great hackers? How do you know when you meet one? That turns out to be very hard. Even hackers can't tell. I'm pretty sure now that my friend Trevor Blackwell is a great hacker. You may have read on Slashdot how he made his own Segway. The remarkable thing about this project was that he wrote all the software in one day (in Python, incidentally).

では、誰が偉大なハッカーなんだろう。そういう人と出会った時に、そのことをどうやって知ればいいのか。実は、これは難しい。ハッカーでさえもわからない。私は、友人のトレバー・ブラックウェルが偉大なハッカーであるということをほぼ確信している。彼が自分でセグウェイを作ってしまった話を、あなたはスラッシュドットで読んだことがあるかもしれない。そのプロジェクトのすごいところは、彼は全てのソフトウェアを一日で書いてしまったということだ(ちなみに彼は **Python** を使っていた)。

For Trevor, that's par for the course. But when I first met him, I thought he was a complete idiot. He was standing in Robert Morris's office babbling at him about something or other, and I remember standing behind him making frantic gestures at Robert to shoo this nut out of his office so we could go to lunch. Robert says he misjudged Trevor at first too. Apparently when Robert first met him, Trevor had just begun a new scheme that involved writing down everything about every aspect of his life on a stack of index cards, which he carried with him everywhere. He'd also just arrived from Canada, and had a strong Canadian accent and a mullet.

トレバーにとっては、それはごく普通のことだった。でも私は彼と初めて会った時、彼のことを間抜けだと思っていたんだ。彼はロバート・モリスのオフィスに突っ立ってて何やらごによごによと話していて、私はトレバーの後ろに立って、身ぶり手ぶりでロバートにこの間抜けをさっさと追い出してランチに行こうぜ、と伝えようとしていた。ロバートも最初はトレバーのことを誤解したと言っている。ロバートが最初に彼と会った時、トレバーは、いつもどこへでも持ち歩いているカードに、彼の人生の全ての面を書き出す、という作業に没頭しはじめたところだった。彼はまたカナダからやってきたばかりで、強いカナダなまりで、髪を長く伸ばしていた。

The problem is compounded by the fact that hackers, despite their reputation for social obliviousness, sometimes put a good deal of effort into seeming smart. When I was in grad school I used to hang around the MIT AI Lab occasionally. It was kind of intimidating at first. Everyone there spoke so fast. But after a while I learned the trick of speaking fast. You don't have to think any faster; just use twice as many words to say everything.

ハッカーが、社交に疎いという評判にもかかわらず、自分を賢く見せようと努力するという事実が、さらに問題を複雑にする。私は大学院にいた頃、よく MIT AI ラボに遊びに行っていたのだが、最初はちょっとおっかなかったものだ。なにしろ皆おそろしいほど速くしゃべっていたからだ。でもしばらくしたら、私も速くしゃべるトリックを身に付けることができた。速く考える必要はないんだ。ただ何を言うにも 2 倍の言葉を使えばいい。

With this amount of noise in the signal, it's hard to tell good hackers when you meet them. I can't tell, even now. You also can't tell from their resumes. It seems like the only way to judge a hacker is to work with him on something.

これだけのノイズにシグナルが埋もれていると、誰かに出会った時にそれが良いハッカーかどうかを知るのはとても難しい。私は、今でさえ、わからない。履歴書を見たって役に立たない。ハッカーを判断する唯一の方法は、何かのプロジェクトと一緒に仕事をしてみ

ることしかないような気がする。

And this is the reason that high-tech areas only happen around universities. The active ingredient here is not so much the professors as the students. Startups grow up around universities because universities bring together promising young people and make them work on the same projects. The smart ones learn who the other smart ones are, and together they cook up new projects of their own.

それが、ハイテク企業の集合が大学の周辺にしか現れない理由かもしれない。教授がたくさんいることよりも、学生がたくさんいることが重要な要素なんだ。ベンチャーが大学の周辺で育つのは、大学が才能ある若い人々を集めて同じプロジェクトで作業させるからだ。優秀な人間は他の優秀な人間から学び、一緒に自分たちの新しいプロジェクトを立ち上げる。

Because you can't tell a great hacker except by working with him, hackers themselves can't tell how good they are. This is true to a degree in most fields. I've found that people who are great at something are not so much convinced of their own greatness as mystified at why everyone else seems so incompetent.

一緒に仕事をしない限り偉大なハッカーかどうか分からない、ということは、ハッカー自身が自分がどれだけ優秀かわからない、ということにもなる。これはどの分野でも、ある程度までは真実だ。何かについて非常に優秀な人は、自分の素晴らしさをそれほど信じていなくて、他の人がなんで同じようにできないのか不思議に思うものだ。

But it's particularly hard for hackers to know how good they are, because it's hard to compare their work. This is easier in most other fields. In the hundred meters, you know in 10 seconds who's fastest. Even in math there seems to be a general consensus about which problems are hard to solve, and what constitutes a good solution. But hacking is like writing. Who can say which of two novels is better? Certainly not the authors.

だがハッカーの場合、どれだけ優秀かを知るのはとりわけ難しい。作品を比較するのが難しいからだ。他の分野ではそれが簡単なことが多い。100メートル走なら、10秒のうちに誰が一番速いかわかる。数学のような世界でも、どの問題が解くのが難しく、どういう解が良い解かということについて一般的な合意があるようだ。でもハッキングは、小説を書くみたいなものだ。2つの小説のどちらが良い、なんて誰が比較できるだろう。少なくとも著者には無理だ。

With hackers, at least, other hackers can tell. That's because, unlike novelists,

hackers collaborate on projects. When you get to hit a few difficult problems over the net at someone, you learn pretty quickly how hard they hit them back. But hackers can't watch themselves at work. So if you ask a great hacker how good he is, he's almost certain to reply, I don't know. He's not just being modest. He really doesn't know.

ハッカーの間では、少なくとも、他のハッカーがそれを知ることができる。小説家と違って、ハッカーはプロジェクトで協力するからだ。ネット越しに誰かにいくつかの難しい問題を投げれば、彼らはたちまち別の問題を投げ返してくるだろう。ハッカーは、しかし、自分自身が仕事をしているところを見ることはできない。だから、偉大なハッカーに、自分はどれくらいすごいと思うかと聞いたら、まずこういう答えが返って来るだろう。分からないよ、と。謙遜しているんじゃない。本当に分からないんだ。

And none of us know, except about people we've actually worked with. Which puts us in a weird situation: we don't know who our heroes should be. The hackers who become famous tend to become famous by random accidents of PR. Occasionally I need to give an example of a great hacker, and I never know who to use. The first names that come to mind always tend to be people I know personally, but it seems lame to use them. So, I think, maybe I should say Richard Stallman, or Linus Torvalds, or Alan Kay, or someone famous like that. But I have no idea if these guys are great hackers. I've never worked with them on anything.

そして、他のハッカーにしたって、自分が直接一緒に仕事をした相手じゃないと分からない。このことから、ハッカー界には奇妙な状況が生じる。我々のヒーローは誰なのか、誰にも分からないんだ。有名になったハッカーというのは、偶然の宣伝の事故みたいなもののせいで有名になることが多いようだ。時々、素晴らしいハッカーの例を挙げる必要に迫られることがあるが、私は誰を挙げたらいいのか、分かったためしがない。すぐに頭に浮かぶのは、個人的に直接知っている人たちだが、なんだかそれを例として挙げるのも不遜なような気がする。それよりは、リチャード・ストールマンだとか、リーナス・トーバルスだとか、アラン・ケイだとか、有名人を挙げるほうが良さそうな気がする。でも、彼らが本当に素晴らしいハッカーなのか、正直なところ私には知る術がない。一緒に何か仕事をしたことがないからだ。

If there is a Michael Jordan of hacking, no one knows, including him.

ハッカー界のマイケル・ジョーダンがいるとしたら、それは彼自身も含めて、誰にも知られることがないだろう。

Cultivation

育てるには

Finally, the question the hackers have all been wondering about: how do you become a great hacker? I don't know if it's possible to make yourself into one. But it's certainly possible to do things that make you stupid, and if you can make yourself stupid, you can probably make yourself smart too.

最後に、ハッカーが皆考えている質問を扱おう。どうやったら素晴らしいハッカーになれる?誰かを素晴らしいハッカーにする方法ってのがあるのかどうか、私は知らない。けれど、これをしていれば自分は間抜けになるだろう、ということは確かにある。自分を間抜けにする方法があるなら、自分を賢くする方法もあるかもしれない。

The key to being a good hacker may be to work on what you like. When I think about the great hackers I know, one thing they have in common is the extreme [difficulty](#) of making them work on anything they don't want to. I don't know if this is cause or effect; it may be both.

良いハッカーになる鍵は、たぶん、自分がやりたいことをやることだ。私が知っている素晴らしいハッカーを考えてみると、彼らに共通することのひとつは、彼らが自分から望まないことをやらせるのは極めて難しいだろうということだ。これが原因なのか、結果なのかは定かではない。もしかすると両方かもしれない。

To do something well you have to [love](#) it. So to the extent you can preserve hacking as something you love, you're likely to do it well. Try to keep the sense of wonder you had about programming at age 14. If you're worried that your current job is rotting your brain, it probably is.

何かをうまくやるためには、それを愛していなければならない。ハッキングがあなたがやりたくてたまらないことである限りは、それがうまくできるようになる可能性が高いだろう。14才の時に感じた、プログラミングに対するセンス・オブ・ワンダーを忘れないようにしよう。今の仕事で脳味噌が腐っていつてるんじゃないかと心配しているとしたら、たぶん腐っているよ。

The best hackers tend to be smart, of course, but that's true in a lot of fields. Is there some quality that's unique to hackers? I asked some friends, and the number one thing they mentioned was curiosity. I'd always supposed that all smart people were curious-- that curiosity was simply the first derivative of knowledge. But apparently hackers are particularly curious, especially about how things work. That makes sense, because programs are in effect giant descriptions

of how things work.

最高のハッカーはもちろん賢いけれど、それは他のいろいろな分野でも同じだ。ハッカーについてだけ特有な資質というのはあるだろうか。何人かの友人に尋ねてみた。最初にあがってくる答えは、好奇心だった。賢い人はみな、好奇心が強いと私は思っている。好奇心は単に知識の一階微分だからだ。それでも、ハッカーはとりわけ好奇心が強いようだ。特に、ものが動く仕組みに関しては。ある意味それは当然かもしれない。プログラムというものの自体、実質的に、ものが動くしくみの巨大な記述だからだ。

Several friends mentioned hackers' ability to concentrate-- their ability, as one put it, to "tune out everything outside their own heads." I've certainly noticed this. And I've heard several hackers say that after drinking even half a beer they can't program at all. So maybe hacking does require some special ability to focus. Perhaps great hackers can load a large amount of context into their head, so that when they look at a line of code, they see not just that line but the whole program around it. John McPhee wrote that Bill Bradley's success as a basketball player was due partly to his extraordinary peripheral vision. "Perfect" eyesight means about 47 degrees of vertical peripheral vision. Bill Bradley had 70; he could see the basket when he was looking at the floor. Maybe great hackers have some similar inborn ability. (I cheat by using a very [dense](#) language, which shrinks the court.)

友人の幾人かは、ハッカーの集中力に関して言及した。一人の言葉を借りれば、「他の全てのことを頭から追い出せる」能力だ。私も確かにそれには気付いていた。また、何人かのハッカーが、ビールを半杯でも飲んだら全くプログラムできなくなると言っているのも聞いたことがある。ハッキングは、ある種の特殊な集中能力を必要とするのかもしれない。素晴らしいハッカーはたぶん、非常に大きなコンテキストを頭の中にロードすることができて、したがってコードを一行眺めている時にも、その行だけでなくそれを取り巻くプログラム全てを見ているのかもしれない。ジョン・マクフィーはビル・ブラッドレイのバスケットボール選手としての成功は、彼の並外れた周辺視に依っていたと書いていた。通常、完全な視力は、だいたい47度の垂直周辺視角を持っている。ビル・ブラッドレイは70度の視角を持っていた。彼は床を見つめていながら、バスケットを観ることができたのだ。素晴らしいハッカーは、そのような生まれつきの能力を持っているのかもしれない(私は密度の高い言語を使うことでずるをしている。それは実質的にコートを狭める効果を持っているからだ)。

This could explain the disconnect over cubicles. Maybe the people in charge of facilities, not having any concentration to shatter, have no idea that working in a

cubicle feels to a hacker like having one's brain in a blender. (Whereas Bill, if the rumors of autism are true, knows all too well.)

このことは、仕切りのブースに対する態度の違いを説明できるかもしれない。オフィスのデザインをする人は、邪魔されたら台無しになるような集中をしないから、仕切りブースで仕事をするのがハッカーにとっては脳味噌をミキサーにかけられるように感じられるなんてことが想像もできないんだ。(一方で、ビルは、彼が自閉症だという噂が本当なら、このことを痛いほど分かっているだろう)。

One difference I've noticed between great hackers and smart people in general is that hackers are more [politically incorrect](#). To the extent there is a secret handshake among good hackers, it's when they know one another well enough to express opinions that would get them stoned to death by the general public. And I can see why political incorrectness would be a useful quality in programming. Programs are very complex and, at least in the hands of good programmers, very fluid. In such situations it's helpful to have a habit of questioning assumptions.

素晴らしいハッカーと、一般的に賢い人々との間の違いとして私が気づいたのは、ハッカーはより「政治的に正しく」ないということだ。良いハッカーの間での、秘密の握手みたいなものがあるくらいだ。ハッカー同士が十分に親しくなると、一般の人が聞いたら嘔然としすぎて死んでしまうくらい恐ろしいことを平気で言い出し始めるのがそれだ。そして、政治的に正しくないことがプログラミングにおいて有用な資質であることも私には分かる。プログラムというものは極めて複雑なもので、しかも少なくとも良いプログラマの手にあるのは、極めて流動的なものだ。そんな状況では、前提を常に疑う習慣は役に立つ。

Can you cultivate these qualities? I don't know. But you can at least not repress them. So here is my best shot at a recipe. If it is possible to make yourself into a great hacker, the way to do it may be to make the following deal with yourself: you never have to work on boring projects (unless your family will starve otherwise), and in return, you'll never allow yourself to do a half-assed job. All the great hackers I know seem to have made that deal, though perhaps none of them had any choice in the matter.

これらの資質を育てることができるだろうか。私には分からない。でも、少なくとも、それらを抑え付けないようにすることはできるだろう。だから、私に言える一番のレシピはこうだ。自分を素晴らしいハッカーにすることができるとしたら、その方法とは、自分自身に対して次の契約を結ぶことだ。以降、退屈なプロジェクトの仕事は一切しなくて良い(家族が餓死しそうでない限りは)、その代わりに、絶対に中途半端な仕事はしない。私が知っている全ての素晴らしいハッカーたちは、皆その契約を結んだように思える。もっとも彼

らにとっては、それはわざわざ選ぶようなことではなかっただろうけれど。

Notes

注釈

[1] In fairness, I have to say that IBM makes decent hardware. I wrote this on an IBM laptop.

[1] 公平のために記しておく、IBM のハードウェアは素晴らしい。私もこの原稿を IBM のラップトップで書いている。

[2] They did turn out to be doomed. They shut down a few months later.

[2] そして、確かに彼らは亡びる運命であった。数ヵ月後に彼らは会社を畳んだのだ。

[3] I think this is what people mean when they talk about the "meaning of life." On the face of it, this seems an odd idea. Life isn't an expression; how could it have meaning? But it can have a quality that feels a lot like meaning. In a project like a compiler, you have to solve a lot of problems, but the problems all fall into a pattern, as in a signal. Whereas when the problems you have to solve are random, they seem like noise.

[3] (これは人々が「人生の意味」について語る時にまさに頭にあることだと私は思う。表面的には、人生の意味ってのは妙な考えだ。人生は数式じゃない。それが意味を持つとは？ 実際は、人生は、意味にとってもよく似たある種の質を持っている。コンパイラのプロジェクトでは、たくさん問題を解くことになるが、それらの問題はひとつのパターンを形作ってゆく。信号のように。一方、解くべき問題がランダムであれば、それはノイズのように知覚されるだろう)。

[4] Einstein at one point worked designing refrigerators. (He had equity.)

[4] アインシュタインは一時期、冷蔵庫を設計する仕事をしていたことがあった(資産を持っていたのにだ)。

[5] It's hard to say exactly what constitutes research in the computer world, but as a first approximation, it's software that doesn't have users.

[5] コンピュータの世界では何をもって研究とするかははっきり言うのは難しいが、一次近似としては、ユーザのいないソフトを書くことだと言える。

I don't think it's publication that makes the best hackers want to work in research

departments. I think it's mainly not having to have a three hour meeting with a product manager about problems integrating the Korean version of Word 13.27 with the talking paperclip.

最高のハッカーが研究部門で働きたがる理由が、論文を出版できるからということだとは思えない。私が思うに、それは多分、Word 13.27 の韓国語バージョンとおしゃべりなクリップを統合するためにプロダクトマネージャと 3 時間にわたるミーティングをしなくて済むからだろう。

[6] Something similar has been happening for a long time in the construction industry. When you had a house built a couple hundred years ago, the local builders built everything in it. But increasingly what builders do is assemble components designed and manufactured by someone else. This has, like the arrival of desktop publishing, given people the freedom to experiment in disastrous ways, but it is certainly more efficient.

[6] 建築業界でも、似たようなことが長い間起きていた。200 年程前は、家を建てるとなったら、その地域の業者が何から何まで全てを作っていたものだ。だが次第に、業者は他の誰かによって設計され製造された部品を組み立てるように変わってきた。この傾向は、ちょうどデスクトップパブリッシングの到来と同様に、人々にひどいものを作る実験を行う自由を与えたが、確かに効率は良い方法だ。

[7] Google is much more dangerous to Microsoft than Netscape was. Probably more dangerous than any other company has ever been. Not least because they're determined to fight. On their job listing page, they say that one of their "core values" is "Don't be evil." From a company selling soybean oil or mining equipment, such a statement would merely be eccentric. But I think all of us in the computer world recognize who that is a declaration of war on.

[7] Google はマイクロソフトにとってネットスケープよりも遥かに危険だろう。おそらく、今までに存在したどの会社よりも脅威なのではないか。とりわけ、Google は戦う気である。彼らの求人広告ページに、彼ら自身の「コア・バリュー」として、「邪悪にはならない(Don't be evil)」という標語が掲げている。大豆油とか切削工具を売っている会社がそう言ったとしたら、単に変わった会社でしかないだろう。しかし、コンピュータの業界にいる人間なら誰でも、それが誰への宣戦布告なのか、たちどころに理解できると思う。

Thanks to Jessica Livingston, Robert Morris, and Sarah Harlin for reading earlier versions of this talk.